USING OBJECT-ORIENTED DATABASE TECHNOLOGY TO DEVELOP A MULTIPLE DOMAIN CAPABILITY FOR DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEMS

THESIS

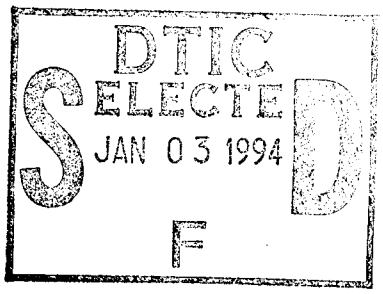
Alfred William Harris, Jr
Captain, USAF

AFIT/GCS/ENG/94D-07

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-07



USING OBJECT-ORIENTED DATABASE TECHNOLOGY TO DEVELOP A MULTIPLE DOMAIN CAPABILITY FOR DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEMS

THESIS

Alfred William Harris, Jr
Captain, USAF

AFIT/GCS/ENG/94D-07

AFIT/GCS/ENG/94D-07

USING OBJECT-ORIENTED DATABASE TECHNOLOGY TO DEVELOP A MULTIPLE DOMAIN CAPABILITY FOR DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEMS

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)

Alfred William Harris, Jr, B.S., M.B.A.

Captain, USAF

December, 1994

Acknowledgements

Many people provided valuable support to me throughout this research effort. First, I'd like to thank my fellow KBSE researchers for having a "team" attitude. Everyone always seemed willing to take valuable time and assist each other with any problems, both simple and complex. Next, I'd like to thank Dr Potoczny and Capt Dan Cecil for serving on my committee. Dan's early efforts in familiarizing me with ITASCA were invaluable. I would also like to thank my thesis advisor, Maj Paul Bailor. Each time I sought advice on an obstacle, he provided the ideas I needed to move forward. Last and most importantly, I'd like to thank my family. Any time I needed to clear my mind, a trip to the park with my children, Laura and Michael, always seemed to work best. My wife Kathy provided more support to me than I could have ever imagined. Her love and understanding allowed me to perform at my best.

Alfred William Harris, Jr

Table of Contents

List of Figures

List of Tables

Table Page

Abstract

This thesis describes the design and implementation of a multiple domain capability for a domain-oriented application composition system, named Architect. The research goal was to show how object-oriented database management system (OODBMS) technology can be used to provide simultaneous access to multiple domain-oriented knowledge bases. Since the Architect system was originally designed using the object-oriented paradigm, insertion of OODBMS technology was relatively simple and many of the object-oriented concepts, such as inheritance and aggregation, proved beneficial. Inheritance was used to encapsulate domain knowledge by defining each domain as a subclass of Architect's software architecture. Aggregation was used to allow applications to cross domain boundaries by nesting components from multiple domains in an application. To validate this approach, domain extensions to two existing domain models were implemented to make the domains compatible in a multiple domain environment, and applications containing objects from both the logic circuits and digital signal processing domains were successfully developed. One of the primary benefits of this research is the potential for greater reuse of objects. To satisfy new requirements, domain engineers can now search for and access objects from other domains as an alternative to implementing them in their own domains.

USING OBJECT-ORIENTED DATABASE TECHNOLOGY TO DEVELOP A MULTIPLE DOMAIN CAPABILITY FOR DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEMS

I. Introduction

1.1 Overview

The primary purpose of this research effort was to extend the object-oriented database management system (OODBMS) support for a prototype domain-oriented application composition system developed by the Knowledge Based Software Engineering research group at AFIT. The focus of the research was to use OODBMS technology to provide a capability for multiple domain applications. The original capabilities of the prototype system limited the composition of applications to include primitive objects from only one domain. However, the capability to compose multiple domain applications adds greater flexibility to the application composition process by allowing domain knowledge to be shared and reused across domain boundaries.

1.2 Background

In previous research, Cecil and Fullenkamp (7) evaluated and implemented the concept of database support for domain-oriented application composition systems. In particular, they applied their research to the Architect system. Architect has been developed by several researchers at AFIT, with the initial work being done by Anderson (2) and

Randour (17). Architect is a prototype system designed to allow a software engineer and an application specialist to work together to compose formally specified software artifacts into formal software system specifications. An application specialist is a highly competent user of the domain-oriented application composition system in a given domain.

Prior to Cecil's and Fullenkamp's work, the persistent technology base of Architect was file-based. By implementing an OODBMS to serve as the technology base, domain data can be persistently stored in an object model form as opposed to being flattened into a file-based format. Figure 1.1 represents the domain-oriented application composition environment developed at AFIT, with the persistent technology base in the lower right corner.

During the development of the OODBMS technology base, Cecil and Fullenkamp needed to develop two object models; one for the application domain and another for the software architecture. These models ultimately determined the structure in which the data was stored in the OODBMS. They used the domain of logic circuits as their validating domain, and thus, built an object model for logic circuits. However, to generalize capturing knowledge for all domains, they developed a meta-model that describes domain model definitions. In effect, the logic circuits object model is an instance of the domain definition meta-model. Since Architect's software architecture is based on the Object-Connection-Update (OCU) model, Cecil and Fullenkamp also developed a software architecture object model that corresponds to the OCU model. These object models guided the implementation of the OODBMS technology base.

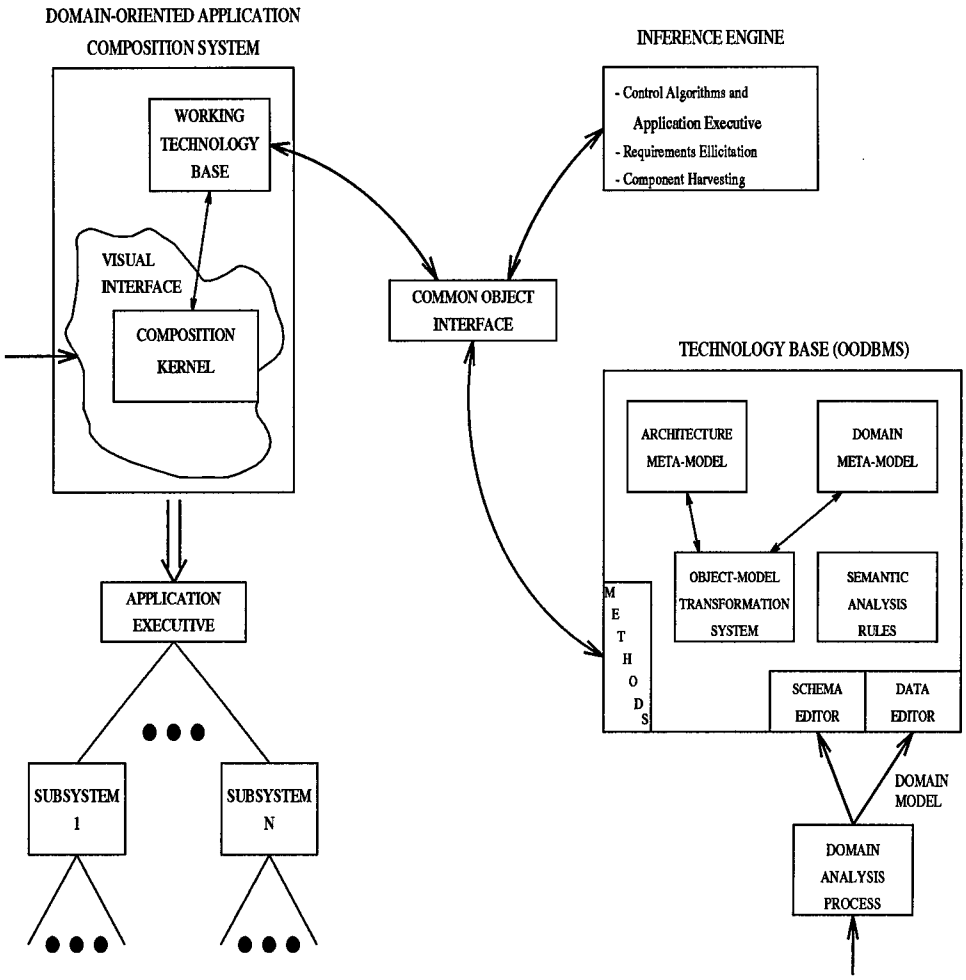


Figure 1.1 Domain-Oriented Application Composition Environment

1.3 Problem

Since the previous research validated the OODBMS technology base for the logic circuits domain only, further research was needed to determine the feasibility of incorporating other domains. The previous designs were made scalable by encapsulating domain and architecture data separately, and relating them through inheritance, as shown in Figure 1.2. For example, the logic circuits domain is modeled as a subclass of the "primitive" object class in the OCU model. In fact, any number of domains can be added to the

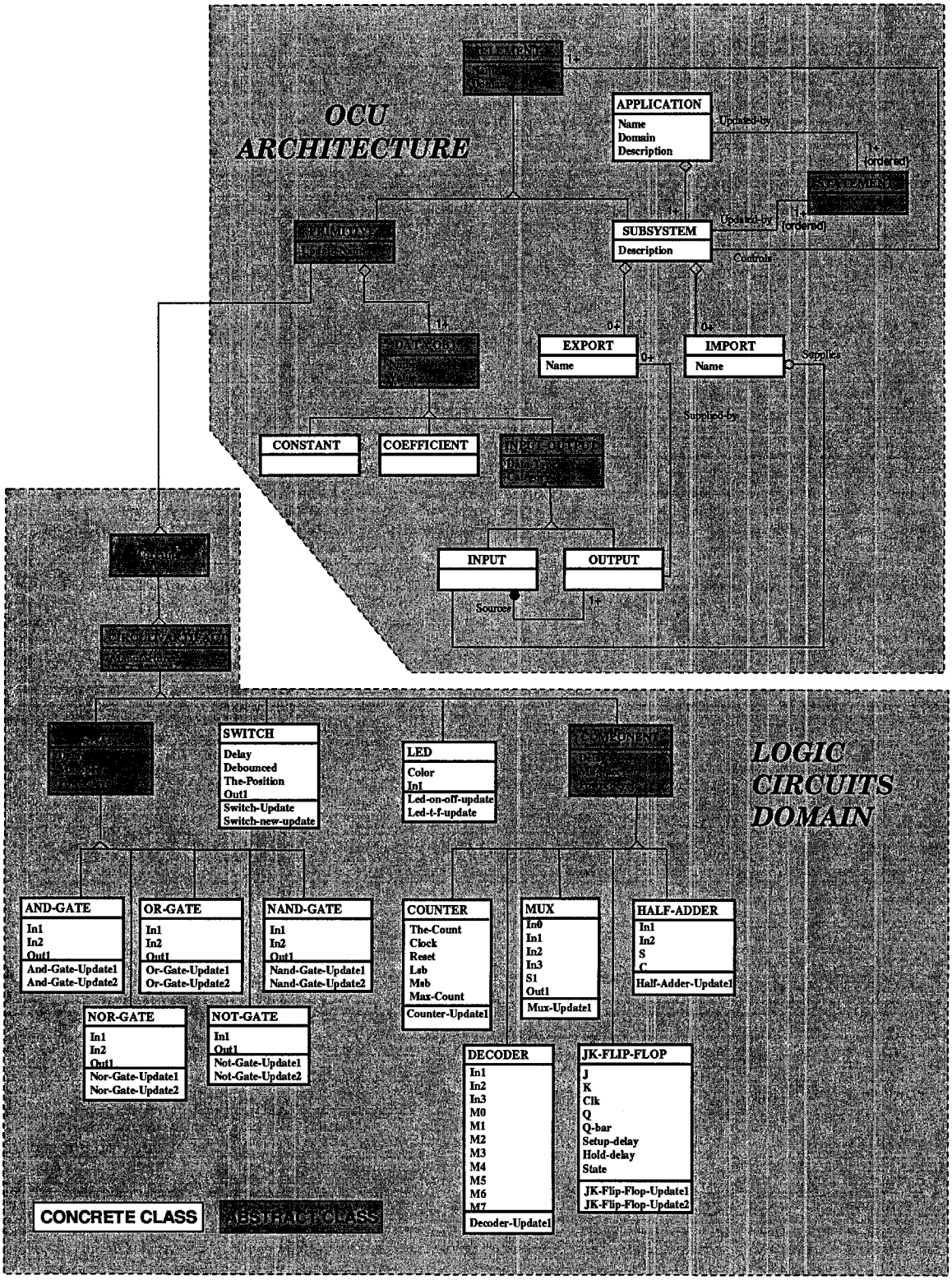


Figure 1.2 Combined Object Models Showing Circuits Inheriting from OCU

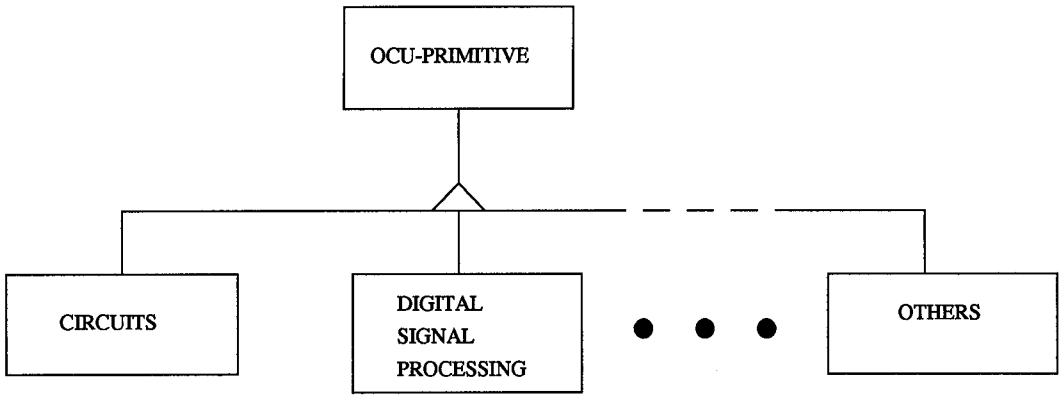


Figure 1.3 Inheritance from OCU by Multiple Domains

database as subclasses of the "primitive" object class. This concept is illustrated in Figure 1.3. However, techniques for sharing components across domains needed to be further investigated.

The capability to build multiple domain applications eliminates the need for expanding domain boundaries in some cases as indicated in the following scenario. When composing an application, a user might need a primitive that is unavailable in a given domain. In this case, the domain engineer needs to expand the domain boundary to include the unavailable primitive. However, if the primitive already exists in another domain, it would be easier to simply use the primitive from the other domain. A multiple domain application capability would provide this benefit. To clearly identify the focus of this research, the following problem statement is offered.

Problem Statement:

Expand existing OODBMS technology base capabilities by incorporating additional application domains and providing the capability to include primitive objects from multiple domains in an application.

1.4 Scope

The purpose of this research did not include modeling and developing knowledge for any particular domain. Other research had already been accomplished in the development of other domains, e.g., digital signal processing. As a result, this effort took advantage of those domain models already developed.

1.5 Assumptions

The ability to compose multiple domain applications is a very beneficial feature based on two assumptions. First, it is conceivable that domain boundaries will not always be clearly defined, that is, an individual primitive might be useful in more than one domain. However, having to implement the primitive in more than one domain is an unnecessary duplication of effort. Second, even when domain boundaries are clearly defined, two domains might naturally interface with each other. Either way, a multiple domain application capability resolves the issue. As explained in Section 1.3, a multiple domain application capability eliminates the need for defining a primitive in more than one domain.

1.6 Approach

The following steps were performed to achieve the capability of composing an application across multiple domains:

- Conduct a literature search to assist in developing the best techniques for sharing components across domain boundaries for a domain-oriented application composition system.

- Identify, analyze, and select application domains suitable for incorporation into the existing OODBMS technology base.
- Design and develop object models for the selected application domains.
- Develop database schemas for the selected application domains.
- Enhance the technology base by implementing chosen domains in the database.
- Analyze the Architect system design and implementation to determine the feasibility of allowing a single application to include primitives from multiple domains.
- Design and implement the changes to the Architect system to allow the composition of applications to cross domains.
- Develop a set of objectives to use in the testing and validation of the enhanced Architect system.
- Execute tests to validate the database and Architect system implementations.

1.7 Summary

This research was a follow-on effort to previous research which applied database technology to support a domain-oriented application composition system called Architect. The focus of this research was to enhance previous efforts by incorporating additional application domains into the Architect technology base and allowing a single application to include primitive objects from multiple domains. This provides the benefits of allowing domain knowledge to be shared and reused across domain boundaries.

1.8 Order of Presentation

The remainder of this thesis is organized as follows. Chapter II is a literature review pertaining to object-oriented database management system support for domain-oriented application composition systems. Chapter III follows with an analysis of a multiple domain capability for Architect applications. Chapter IV explains the design and implementation used to create a multiple domain capability for Architect applications. Chapter V describes the testing and validation of the multiple domain application environment. Finally, Chapter VI contains the conclusions from this research and identifies recommendations for future research.

This thesis also contains three appendices. Appendix A provides a sample script used to generate a multiple domain application on the Architect system. This research required new primitives to be implemented in Architect's technology base. Appendix B contains the code used to implement them. The digital signal processing domain was implemented in the database. Appendix C contains the object model diagrams for the domain.

II. Literature Review

2.1 Introduction

This literature review pertains to object-oriented database management system support for domain-oriented application composition systems. Object-oriented database management systems (OODBMSs) are well suited for many scientific application areas such as software engineering. When compared to traditional technologies such as relational and hierarchical databases, OODBMSs provide many advantages for complex applications. Some of these advantages are identified in this chapter. Since OODBMS technology is relatively immature, much research on the subject is ongoing (12:42).

In this chapter, a high-level overview of the Architect system described in Chapter I is presented. In addition, the rationale supporting the selection of ITASCA as the OODBMS to integrate with Architect is presented. Next, several fundamental concepts of OODBMSs, as applicable to the Architect system, are discussed. The concepts of objects, identity, aggregation, and inheritance are a few of the more important ones pertaining to this research effort. Occasionally, comparisons of OODBMSs versus relational databases are used to illustrate the merits of OODBMSs. Finally, the design of the software architecture used to implement the Architect system is discussed.

2.2 Overview of the AFIT Architect System

As stated in Chapter I, Architect is a software application composition system developed by the Knowledge Based Software Engineering research group at AFIT. Architect allows an application specialist in a given domain to build software applications without

actually writing any software. The application specialist builds the software application by specializing primitive objects stored in Architect's technology base. The organization of these objects must be in accordance with a set of composition rules specified by a software architecture (7).

To illustrate, consider the logic circuits domain supported by Architect and its OODBMS. Real-world logic circuit components such as and-gates, or-gates, switches, etc., are modeled as primitive objects in Architect. Rules of composition define how the various primitive objects can be interconnected. With a graphical user interface, the application specialist can build complex circuits using the primitive objects in accordance with the rules of composition. As a result, the application specialist can generate a software application simulating a circuit such as a full-adder without actually writing the software.

Why is OODBMS support needed for Architect? Architect needs a means of persistent storage for all of the software artifacts it uses and generates. The OODBMS serves as the central repository for those software artifacts. More importantly, for this research, OODBMS technology was crucial in providing a capability to compose applications across domain boundaries. The original Architect system was limited to composing an application within a single domain (4).

2.3 Selection of ITASCA

Before implementing database support for the Architect system, Cecil and Fullenkamp evaluated three OODBMSs: MATISSE, OBJECTSTORE, and ITASCA. They selected ITASCA for two primary reasons. First, ITASCA supports dynamic schema evolution without re-compilation of methods. They felt this feature met their needs for a rapid

prototyping capability (7:3-16). Second, Architect was implemented with SOFTWARE REFINERY™, a formal-based specification and programming environment (7:3-9). SOFTWARE REFINERY and ITASCA both run in the Common Lisp environment. ITASCA provides a remote Lisp interface which makes it convenient for the Architect system to interface to the ITASCA database system.

2.4 Fundamental Concepts of Object-Oriented Database Management Systems

Several fundamental concepts characterize OODBMSs. The concepts of objects, identity, aggregation, and inheritance are a few of the more important ones pertaining to this research effort. These concepts give OODBMSs more powerful modeling capabilities than found in relational database systems (12:45).

2.4.1 Objects. Object-oriented design is centered around the notion of an object. This provides the advantage of allowing the database designer to abstract the problem and solution space in terms of real-world entities. An object is defined in terms of its attributes (5:34). For example, a person could be modeled in a database as an object with the attributes of "name," "sex," "social security number," etc. In contrast, with a relational database the design is centered around tables. Information on a person would be stored in a table in record format (3:44). Each row in the table would have some number of fields, one field for each attribute. In essence, each relational table corresponds to an object class, with each field in the table corresponding to an object attribute.

2.4.2 Identity. The concept of identity means the OODBMS provides a unique identifier for each instance of an object that exists. The identifier is simply a pointer to the

object instance allowing access to the object for data retrieval and manipulation (6:85). In contrast, a relational database does not provide a unique identifier for its entities. The entities in a relational database can only be retrieved and manipulated by their values (1:30). For example, a social security number could serve as an identifier for a "person table." The database designer must build identifiers into the tables of a relational system, whereas an OODBMS automatically provides the identifier.

2.4.3 Aggregation. OODBMSs support the concept of object aggregation. With aggregation, objects are nested within other objects (5:35). For example, an automobile could be modeled as an object with the attributes of "identification number," "color," "engine," etc. However, these attributes do not have to be declared as primitive data types. Instead, they can be declared as object classes, and this is what brings about aggregation. Continuing with the example, the "engine" attribute could be declared as an object containing the attributes of "displacement," "number of cylinders," etc. As a result, the engine is nested within the automobile. The levels of object nesting is not limited.

Relational databases do not support aggregation. This is because the schemas in relational databases generally comply with first normal form. First normal form requires each field (corresponding to an attribute) in a relational table to be atomic, or of a primitive data type such as a number or character string. When all fields of a relational schema are atomic, the schema is said to be in first normal form (13:209). Since each attribute must be of a primitive data type, "objects" can not be nested within other "objects" in a relational table.

2.4.4 Inheritance. Inheritance is the last OODBMS concept discussed here. Inheritance is essentially a reuse mechanism. Inheritance allows the extension or specialization of existing classes by adding additional attributes (3:44). To illustrate, consider the "person" object described earlier. Next, suppose a "student" object was needed in the database. An OODBMS allows the student to inherit all the attributes of the person such as "name," "sex," etc., by simply defining the "student" object as a subclass of the "person" object. Then, additional attributes such as "grade point average" can be added to the "student" object. In contrast, relational databases do not support inheritance.

2.5 Other Object-Oriented Database Management System Considerations

2.5.1 Dynamic Schema Evolution. Much of the work done in software engineering, and in particular on the Architect system, is of a rapid prototyping nature. The work environment is exploratory and evolutionary. Some OODBMSs provide features that support a rapid prototyping environment. One of these helpful features is dynamic schema evolution.

The database schema defines the structure of the data to be stored in the database. In a rapid prototyping environment, such as ours, schema changes are frequent and must be supported with minimal slowdown. The ability to modify relational schemas is limited (5:45). However, some OODBMSs allow schema modification to occur at runtime, instead of requiring a system shutdown (1:36). This is helpful in a rapid prototyping environment as objects tend to evolve rapidly. As mentioned in Section 2.3, one of the reasons Cecil and Fullenkamp selected ITASCA was because of its ability to support dynamic schema evolution.

2.5.2 Applicability of an OODBMS for the Architect System. All of the domains supported by Architect are modeled using the object-oriented paradigm (7:1-4). As a result, the software artifacts used and generated by Architect can be cleanly mapped to an OODBMS. OODBMS support for the Architect system can be further justified by examination of some of the object models developed by Cecil and Fullenkamp. The object diagrams in their thesis adhere to the Object Modeling Technique as described in Rumbaugh's book, *Object-Oriented Modeling and Design* (21). The diagrams in this thesis also adhere to the Object Modeling Technique unless indicated otherwise.

Cecil and Fullenkamp used the logic circuits domain in their efforts to provide OODBMS support for the Architect system. They made heavy use of inheritance in their object model for the logic circuits domain, as shown in Figure 2.1. For example, the "gate" object class has the attributes of "delay," "mil-spec?," and "power level." The "gate" object is a superclass for five subclasses: "and-gate," "or-gate," "nand-gate," "nor-gate," and "not-gate." Each of the subclasses inherits the attributes of "delay," "mil-spec?," and "power level" from the superclass. Another example of inheritance within the logic circuits domain is the "component" object class with its five subclasses: "counter," "mux," "half-adder," "decoder," and "JK flip-flop." With ITASCA, Cecil and Fullenkamp were able to build schemas taking advantage of these inheritance associations.

Cecil and Fullenkamp also developed a meta-model for domain models. The meta-model is a model from which all domain models can be built. In this regard, the logic circuits domain model described earlier can be thought of as an instance of the meta-model, shown in Figure 2.2. The meta-model makes heavy use of aggregation. A domain definition is composed of one or more object classes. Each object class is composed of zero or more

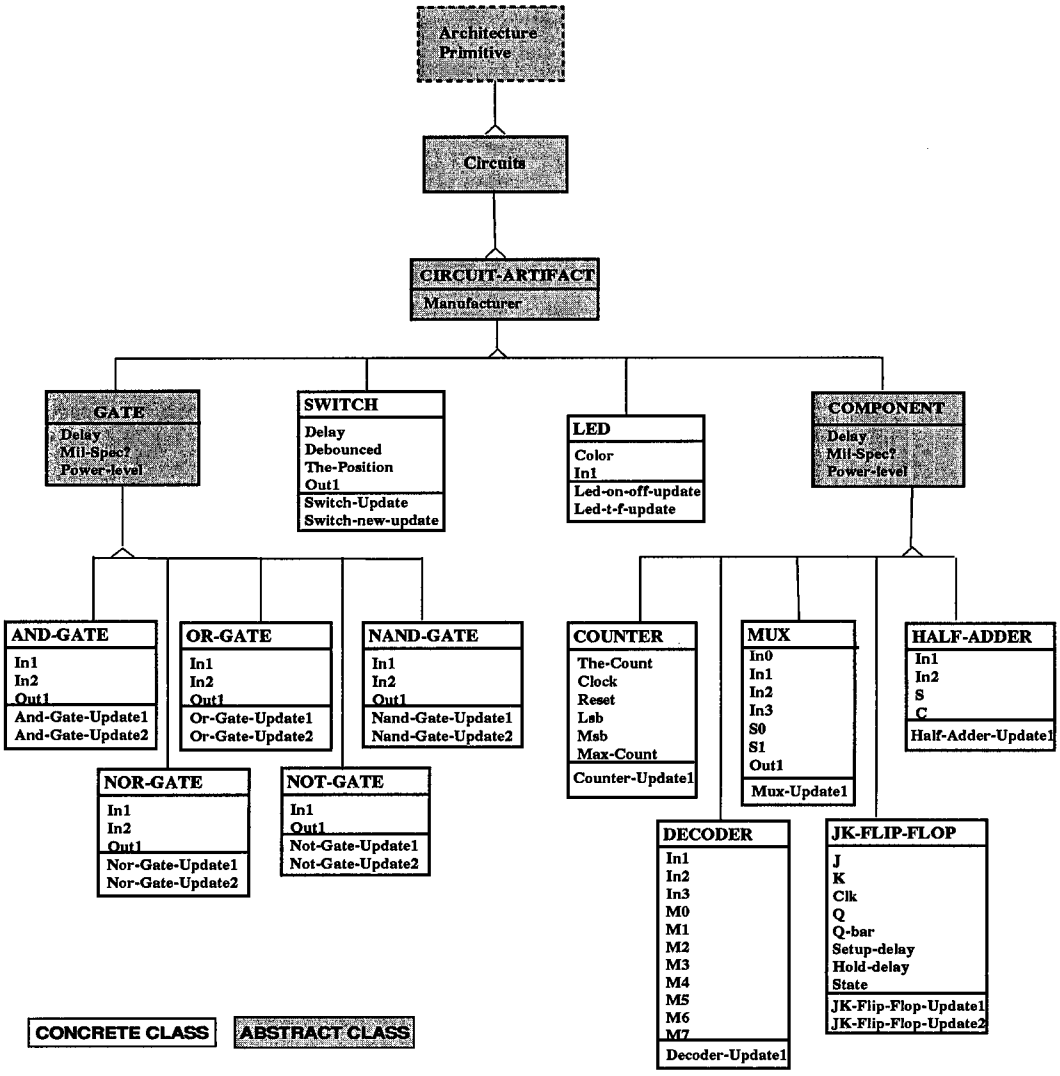


Figure 2.1 Object Model of Logic Circuits Domain

data objects. Finally, each concrete object class contains one or more REFINE functions. With ITASCA, Cecil and Fullenkamp were able to build schemas taking advantage of these aggregate associations.

2.6 Software Architectures

The software architecture becomes a major design consideration of a software system as its size and complexity increase. A higher level of abstraction is used in the design of the

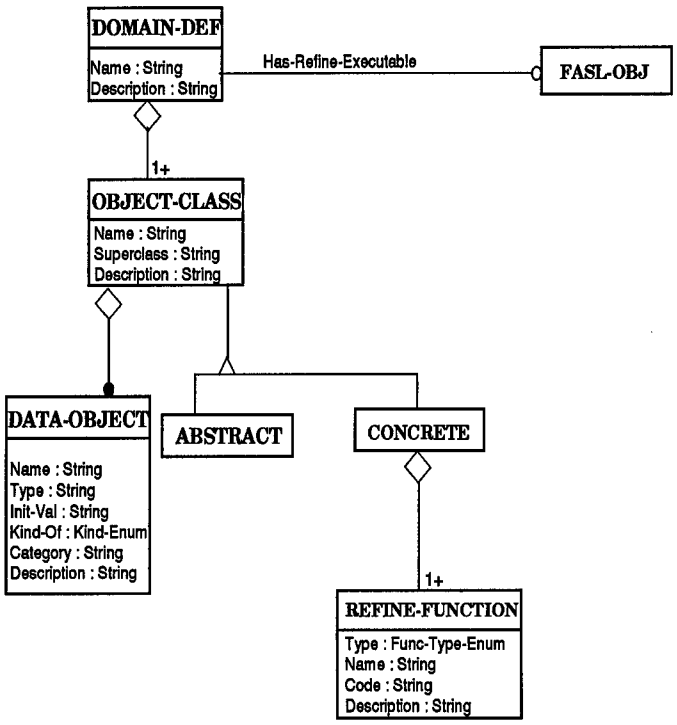


Figure 2.2 Domain-Definition Object Model

software architecture when compared to the design of algorithms and data structures. This is because software architecture is concerned with the overall organization of the system (10), whereas individual algorithms and data structures impact parts of the overall system.

The software architecture for the Architect system was built using the Object-Connection-Update (OCU) model. Software systems in compliance with the OCU model are composed of a group of communicating subsystems. A diagram of an OCU subsystem is shown in Figure 2.3. The controller manages control between a set of objects based on the subsystem's mission. The objects represent real-world or virtual components. The import area is the focal point for the subsystem to gain access to external data. Conversely, the

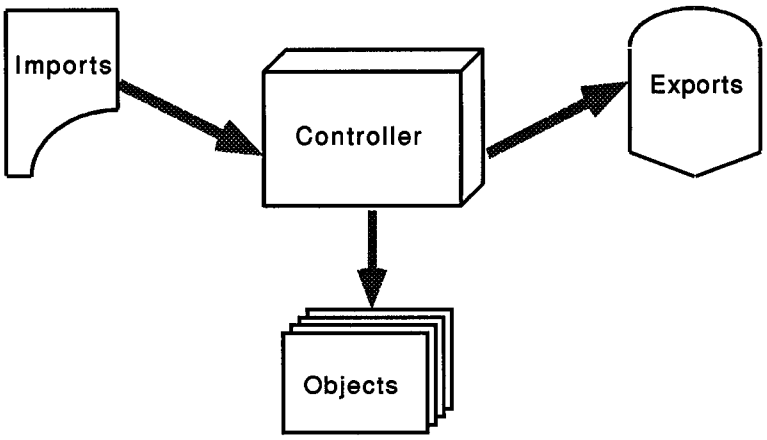


Figure 2.3 OCU Subsystem Construction

export area is the focal point to make internal data available to the environment outside the subsystem (14:17-19).

The software architecture used to implement the Architect system has a direct impact on Architect's database schema design. The object model Cecil and Fullenkamp developed for the software architecture precisely defines the organization used to store many of Architect's software artifacts in the database. Further information on the OCU architecture and alternative software architectures for domain-oriented application composition systems can be obtained from Gool's thesis (11).

2.7 Summary

Object-oriented database management systems provide better support than traditional database systems for some scientific application areas. The fundamental concepts of objects, identity, aggregation, and inheritance give OODBMSs more modeling power than found in older database technologies. Discussion in subsequent Chapters identifies how these powerful modeling capabilities were crucial to solving the problem statement of

this thesis. In particular, the concepts of aggregation and inheritance allow much greater flexibility in the storage and retrieval of applications for Architect than is capable with the original file-based system. This flexibility allowed for the aggregation of primitives from multiple domains in an application.

In addition, this chapter identified that OODBMSs provide support for a rapid prototyping environment with dynamic schema evolution. Finally, the software architecture design used for the Architect system was briefly discussed.

III. Analysis of Multiple Domain Application Capability for Architect

3.1 Introduction

In this chapter, an assessment of Architect's baseline operational capabilities is presented. This leads to a discussion of the desired operational capabilities required to support multiple domain applications. Next, Architect's software environment is analyzed to explain the single domain limitation for applications in the baseline system. The software environment plays a different role in the original, file-based version of Architect than it does in the database version of Architect; these differences are analyzed. Domains participating in a multiple domain application must be compatible. A discussion of Architect's semantic checks provides insight into this compatibility requirement. Since the baseline database version of Architect has only one domain implemented, additional domains must be incorporated. Thus, the meta-model for domain definitions is analyzed. Finally, object-oriented database techniques for sharing components across domain boundaries are discussed.

3.2 Architect Operational Capabilities

To fully understand the problem of providing a multiple domain application capability for Architect, the pertinent operational capabilities of the system need to be understood. A discussion of the baseline and proposed capabilities follows.

3.2.1 Baseline Capabilities. The baseline Architect system is available in either the database version developed by Cecil and Fullenkamp, or the original file-based version. In either version, Architect allows a user to create, edit, save, load, and execute applications within a single domain.

Once the Architect system is loaded, the Architect Visual System Interface (AVSI) is presented to the user. AVSI is the graphical user interface developed by previous researchers, Weide (24) and Cossentine (8). The user may then interact with the system to perform operations on an application. The first operation usually performed is either "Create New Application" or "Load Saved Application." For these operations, Architect immediately prompts the user to specify a domain. If creating a new application, the user can begin composing the application and must include at least one subsystem. Eventually, the user reaches the "edit subsystem" phase. In this phase, the user is presented with a window containing an image of the OCU subsystem model. This window is shown in Figure 3.1.

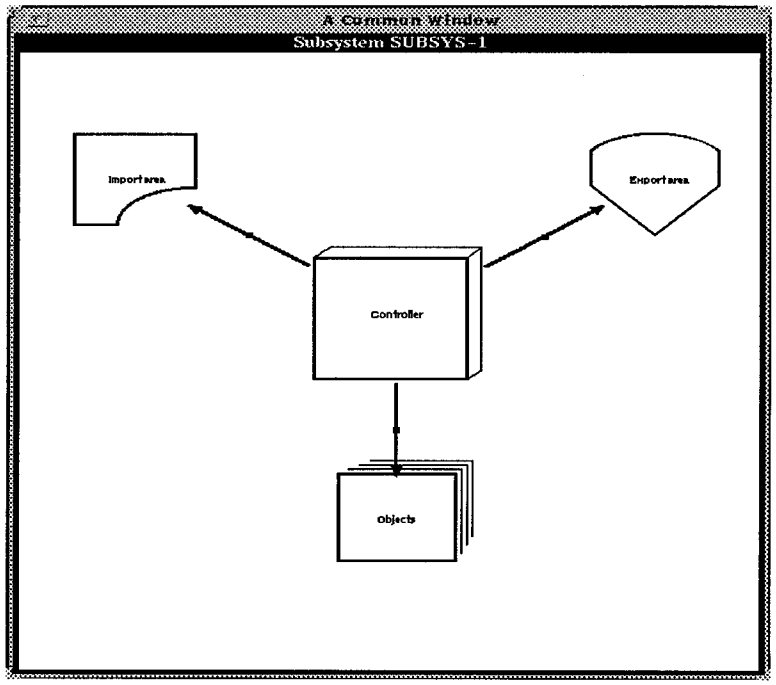


Figure 3.1 Edit Subsystem Window

Next, the user can perform a mouse operation to click on the "objects" icon of Figure 3.1, causing two windows to appear. One of these windows contains all the primitives belonging to the domain of the current application. The technology base window for the logic circuits domain is shown in Figure 3.2. The second window is the subsystem window

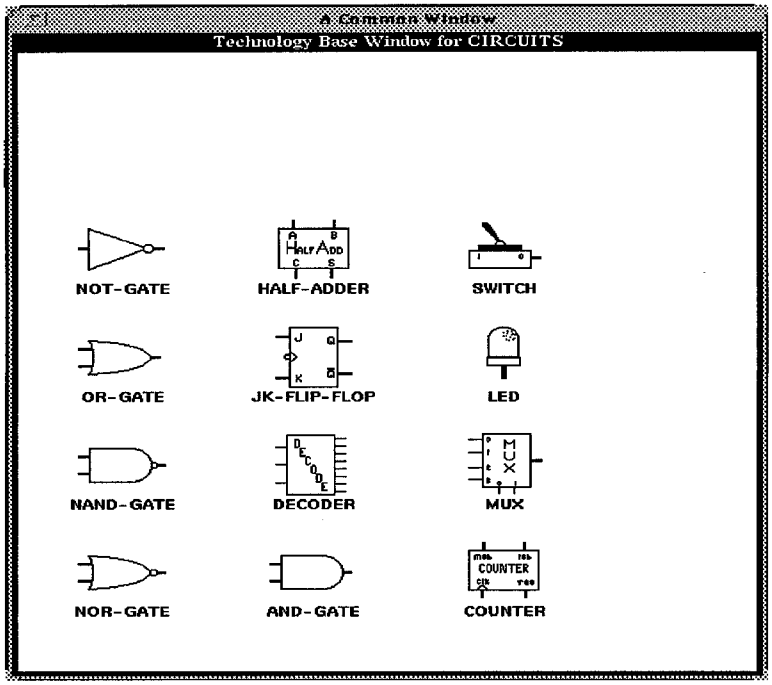


Figure 3.2 Technology Base Window for CIRCUITS

where the user composes the subsystem by placing instances of all required primitive objects into the window. The user is limited to choosing objects from the domain presented in the technology base window. Of course, this is to be expected since applications can contain primitives from only one domain. The user can place objects into the current subsystem by simply performing a mouse operation to drag instances of primitives from the technology base window into the subsystem window. An example of a subsystem window containing one "switch" instance and one "LED" instance is shown in Figure 3.3. Finally,

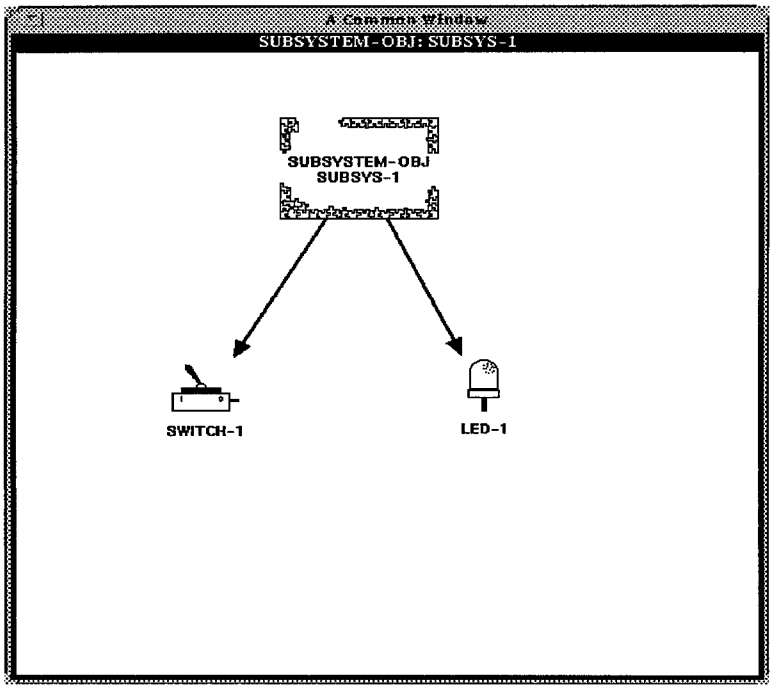


Figure 3.3 Subsystem Window

after all the desired primitive objects have been placed into the subsystem window, the user can deactivate the subsystem and technology base windows and move on to other functions.

3.2.2 Proposed Capabilities. If a multiple domain application capability is to be realized, the user needs an added option to select primitive objects from a domain other than the one originally specified when creating a new application or loading a saved application. This can be accommodated by enhancing the menu options AVSI provides. Two alternatives follow.

- OPTION 1: This approach requires the user to specify all the required domains at the very beginning of the application composition process. When a user initially creates a new application, AVSI prompts the user for the domain of the application. This

would be considered the *primary* domain. AVSI could be enhanced to ask a follow-up question to determine if primitives from *alternate* domains are needed.[1] With this information, AVSI could build a technology base window to include primitives from the alternate domains in addition to primitives of the primary domain. Then, when the user clicks on the "objects" icon of Figure 3.1, primitive objects from all the required domains would be displayed.

- OPTION 2: This approach defers the choice of selecting primitives from alternate domains until the "edit subsystem" phase. When the user clicks on the "objects" icon of Figure 3.1, primitives from the primary domain would be presented in the normal fashion. The user would then place all the required primitives of the primary domain into the subsystem window of Figure 3.3. If primitive objects are needed from an alternate domain(s), the user could simply bring up a menu to make the request. This menu would be presented after clicking the mouse in the background of the window of Figure 3.2. After specifying the alternate domain, a new technology base window would appear allowing the user to include the needed primitives. At this point, the application composition process would proceed as usual.

Option 2 was chosen as the preferred alternative for two reasons. First, Option 1 assumes the user knows all the required primitive objects and their respective domains at the beginning of the application composition process. Option 2 provides the user the flexibility to include primitives from all available domains throughout the entire composition process. Second, Option 1 presents the user a single technology base of primitive

[1] Use of the terms *primary* and *alternate* will be used from this point forward when referring to the domains of a multiple domain application.

objects from the primary domain and alternate domains. The potentially large number of primitive objects would make this window difficult to display in a pleasing manner. At any given instance, Option 2 presents a technology base of primitives from only one domain.

The flowcharts in Figure 3.4 summarize the differences between the baseline and pro-

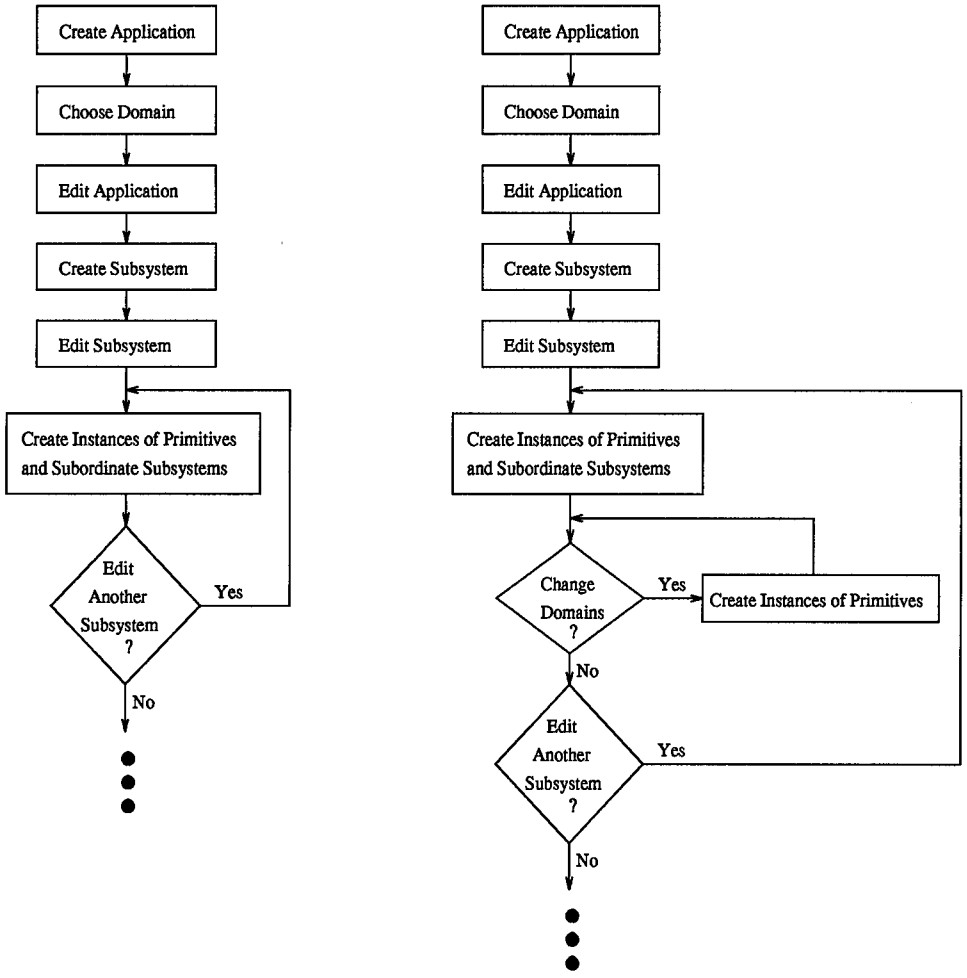


Figure 3.4 Baseline Versus Proposed Capabilities

posed capabilities for Architect's application composition process. The baseline capabilities are shown on the left, and the proposed capabilities are shown on the right.

3.3 Analysis of Architect's Single Domain Limitation

3.3.1 Software Environment. As mentioned in Section 2.3, Architect was implemented with SOFTWARE REFINERY™, a formal-based specification and programming environment. To begin an Architect session, the SOFTWARE REFINERY environment must first be loaded. The SOFTWARE REFINERY environment includes the products REFINE, DIALECT, and INTERVISTA.

- REFINE is a specification and object base manipulation environment. It provides a wide-spectrum language and a structured object base capability (19). The Architect source code is written in the REFINE language.
- DIALECT is used for manipulating formal languages (18:1-1). DIALECT was used to specify Architect's domain specific and architecture (OCU) grammars. DIALECT generates the parser which is used to transform saved applications in textual form to and from the REFINE object base.
- INTERVISTA provides the tools necessary to create interactive user interfaces for REFINE applications. These interfaces include diagrams, pop-up menus, and mouse-sensitive text windows (20:1-1).

3.3.2 File-Based Version of Architect. The Architect limitation for single domain applications is related to the nature of DIALECT's grammar inheritance capabilities. DIALECT allows the management of separate grammars to be handled efficiently with inheritance. This approach uses a common base grammar along with variant grammars to express the differences. Each of the variant grammars inherits the vocabulary and produc-

tions from the base grammar. However, at any given instance, a grammar can only inherit from at most one other grammar (18:5-20).

Architect uses DIALECT's grammar inheritance capabilities for its domain specific grammars. Architect has a general grammar associated with the OCU architecture. Then, there is a domain specific grammar associated with each domain, such as logic circuits, digital signal processing, etc. These domain specific grammars inherit from the general software architecture grammar. As a result, each domain is associated with a grammar that contains the vocabulary and production rules for the OCU architecture, in addition to the vocabulary and production rules for the specific domain. This is beneficial because each grammar contains the knowledge it needs for the software architecture. The disadvantage is that all of the domain knowledge in the grammars is disjoint. In other words, the domain knowledge contained in any given domain specific grammar is not contained in any of the other domain specific grammars. Therefore, any given grammar that is invoked to parse an Architect application contains the vocabulary and production rules of only one domain. Given the current design of the Architect system, DIALECT can not create a parser powerful enough to save and load multiple domain applications.

3.3.3 Database Version of Architect. The database version of Architect offers the same functionality as the file-based version of Architect. However, when applications are saved to the database or loaded from the database, DIALECT's parsing capabilities are not used. Instead, a group of transformation functions developed by Cecil and Fullenkamp are used. When saving an application to the database, the transformation functions take the current Architect application in the REFINE object base, and then store an equivalent

representation of the application in the ITASCA database. Likewise, when loading a saved application from the database, the transformation functions take the database representation of the application and create an equivalent representation in the REFINE object base.

Since the database version of Architect does not use DIALECT's parsing capabilities to save or load applications, the feasibility of multiple domain applications becomes more focused. In fact, the entire hypothesis of using database technology to bring about multiple domain applications for the Architect system is based on the fact that parsing textual representations of applications in and out of the REFINE object base is not required in the database version.

Incidentally, the baseline database version of Architect also allows the user to save or load applications to or from a file just like the file-based version. When performing a save or load of a file in the database version, parsing is invoked in the usual manner. As such, the database version only adds to the user's options; it does not remove any options. However, this is not the case for multiple domain applications. They can only be saved or loaded with the database.

3.4 Compatibility of Domains

The domains participating in a multiple domain application must have some degree of compatibility. At least one primitive from the primary domain must be capable of interfacing with a primitive from an alternate domain. More precisely, primitives from different domains must be able to import and export data between each other.

The Architect system allows the exchange of data between primitives to occur only if semantic checks have been passed. To pass semantic checks, the import and export areas must be of the same type and category. For example, in the logic circuits domain, a switch can export data to the import area of a light. In this case, the type is "boolean" and the category is "signal" for both the export and import areas. If semantics checks fail, a fatal error results.

3.5 Analysis of Meta-Model for Domain Definitions

After Cecil and Fullenkamp completed their research, the database version of Architect was populated with one domain—logic circuits. Obviously, one of the first actions required to bring about a multiple domain application capability was to load another domain in the database. This required an analysis of Cecil's and Fullenkamp's meta-model for domain definitions. The object diagram for the meta-model was shown earlier in Figure 2.2 and is shown again in Figure 3.5 for convenience. The object diagram for the logic circuits domain was shown in Figure 2.1 and is shown again in Figure 3.6 for convenience. The meta-model was used to develop the object diagram for the logic circuits domain. Ultimately, the schema in the ITASCA database was implemented in accordance with the logic circuits domain model.

3.5.1 Primitive Objects and their Attributes. One desired result of developing a domain definition is to provide a technology base of one or more primitive objects which an Architect user can choose from when composing an application. This is achieved in the meta-model by the association that requires a domain definition to be composed of

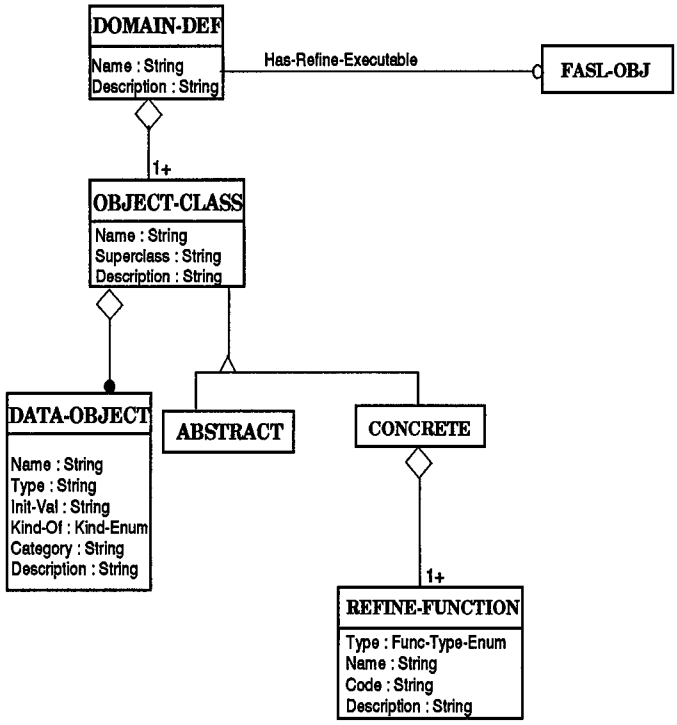


Figure 3.5 Domain-Definition Object Model

one or more object classes. Each primitive object in a technology base corresponds to an instance of the "Object-Class" class in the meta-model. Each primitive object may have any number of attributes associated with it. This is achieved in the meta-model by the association that allows each object class to have any number data objects. The "Data-Object" class in the meta-model corresponds to an object attribute.

To illustrate, consider the following example. One instance of an object class in the logic circuits domain is the "LED" primitive object class. One instance of a data object is the "color" attribute for the "LED" primitive object class.

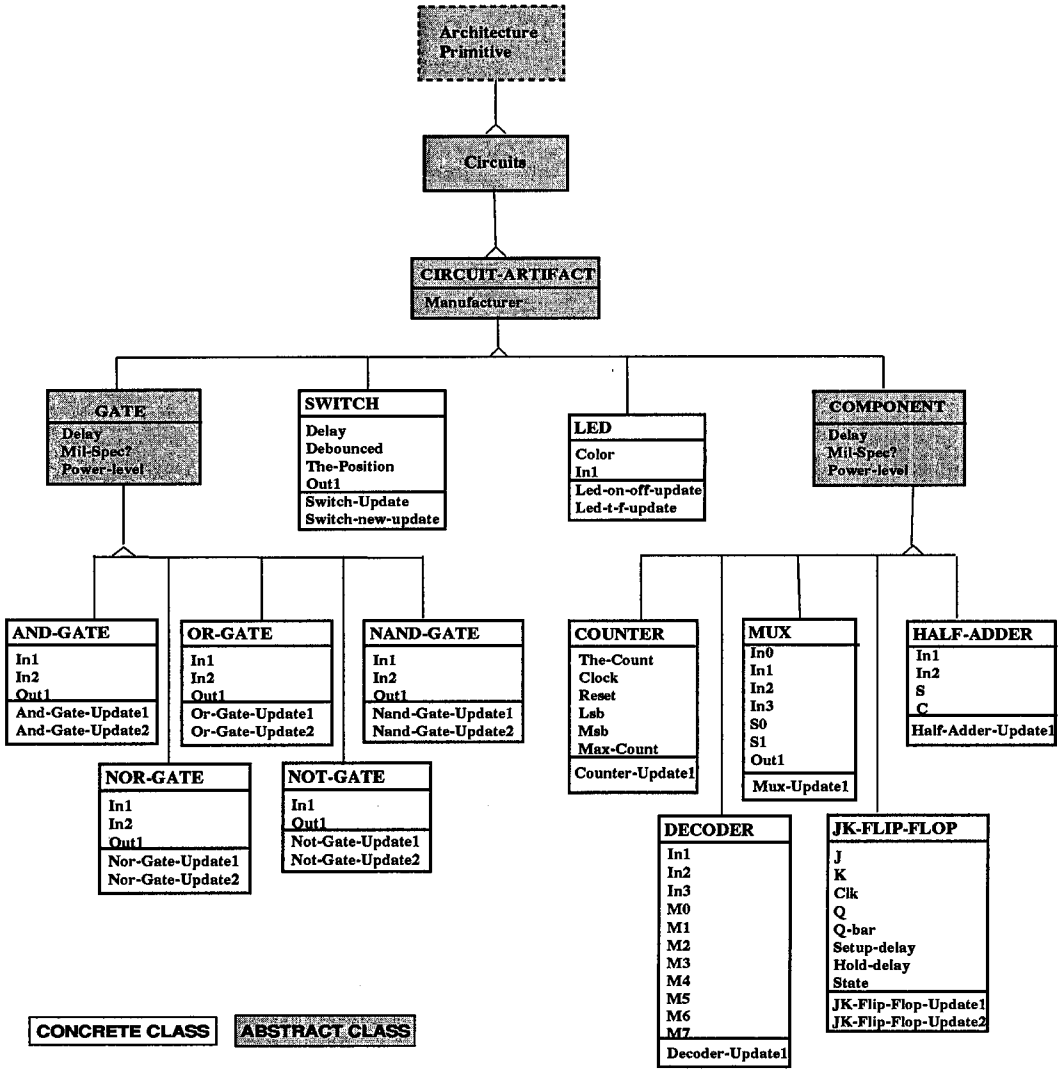


Figure 3.6 Object Model of Logic Circuits Domain

3.5.2 Hierarchy Among Object Classes. In providing a technology base of primitive objects, Cecil and Fullenkamp could have chosen to model each domain with a flat structure. With a flat structure, each primitive object would be positioned one level below the top-level domain definition class. However, they chose to introduce hierarchy among object classes as Figure 3.6 illustrates. This is achieved with the "superclass" attribute of an object class in the meta-model. Each instance of an object class is required to have a superclass, whether it is the top-level domain definition class or some other object

class below the domain definition class. For example, the "LED" object class has the "CIRCUIT-ARTIFACT" object class as its superclass.

The primary benefit of having hierarchy among object classes is to allow multiple levels of inheritance in the ITASCA database. Continuing with the previous example, the "LED" object class of Figure 3.6 inherits the "manufacturer" attribute from the "CIRCUIT-ARTIFACT" object class, as do all the subclasses of the "CIRCUIT-ARTIFACT" object class. Of course, the "LED" object class represents a primitive object in the technology base, while the "CIRCUIT-ARTIFACT" object class does not. The meta-model accounts for this by distinguishing each object class as either abstract or concrete. Each object class representing a primitive object in the technology base is concrete while all other object classes are abstract.

3.5.3 Primitive Object Update Functions. The Architect system requires each primitive object to have an update function. When an application is executed, the application executive ensures each subsystem in the application is updated in the order specified by the application's update algorithm. Likewise, the update function for each object in a given subsystem is executed in an order specified by the subsystem's update algorithm. To support execution of an update function for a primitive object, data can be imported from outside the primitive. Then, the state of the primitive is updated based on the imported data and the primitive's previous state. Finally, data can be made available for export outside the primitive. The meta-model accounts for the primitive update process by requiring each concrete object class to contain at least one REFINE (update) function, as shown in Figure 3.5.

3.5.4 REFINE *Executable.* Each domain definition must have a REFINE "FASL-Object" associated with it in order to make the domain executable in REFINE and Architect. A "FASL-Object" corresponds to the executable file generated by a REFINE compilation. The source code used to generate the "FASL-Object" is automatically created by an ITASCA method named "make-refine-file," written by Cecil and Fullenkamp. After all the domain definition data is loaded into the ITASCA database, the "make-refine-file" method can be executed to generate the source code for the domain. The meta-model allows a domain definition to have a FASL-Object with the "Has-Refine-Executable" association, as Figure 3.5 illustrates.

3.5.5 Other Considerations. Because Architect is a visual system, an icon bitmap must be developed for each primitive to fully define a domain. The icon bitmap information for each primitive object is included in the REFINE source file for the domain. This allows AVSI to generate the technology base window during the composition of an application.

The last consideration here is the ITASCA schema, which enables the database to store instances of Architect applications, including all subsystems and primitive objects contained in those applications. Each domain definition developed from the meta-model must have a corresponding schema implemented in the database. Each object class identified in a given domain must have a corresponding object class built in that domain's database schema. Also, each attribute must be included in the appropriate object class of the schema.

3.6 Database Techniques for Sharing Components Across Domain Boundaries

As stated in Section 3.5.5, each domain supported by the database version of Architect must have a schema implemented in the database to allow for storage of applications. As discussed in Chapter I and illustrated in Figure 1.3, each domain is defined as a separate subclass of the "OCU-Primitive" object class. However, to support multiple domain applications, techniques for allowing a single application to include components across domain boundaries must be analyzed. An analysis of the "OCU-Application," "OCU-Subsystem," and "OCU-Primitive" database object classes provides insight for sharing components across domain boundaries.

3.6.1 "OCU-Application" Object Class. An Architect application is modeled as an object class in the database. The object class is named "OCU-Application" and is shown with its attributes in Figure 3.7. An application has a "Domain" attribute, which

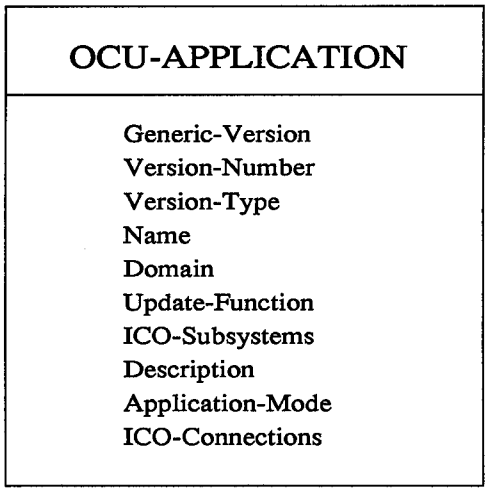


Figure 3.7 Object Model of an OCU Application

of course, identifies the domain from which the application was composed. In a multiple

domain application, the "Domain" attribute refers to the primary domain from which the application was composed; the use of alternate domains is not indicated by this attribute.

The "ICO-Subsystems" attribute means "is composed of subsystems." This attribute makes use of aggregation as described in Section 2.4.3, since its type declaration is a set of "OCU-Subsystem" objects. In other words, "OCU-Subsystem" objects are nested within "OCU-Application" objects. This leads to an analysis of the "OCU-Subsystem" object class to provide an understanding of how an "OCU-Application" object can contain primitive objects from more than one domain.

3.6.2 "OCU-Subsystem" Object Class. A subsystem is also modeled as an object class in the database. The object class is named "OCU-Subsystem" and is shown with its attributes in Figure 3.8. As with the "OCU-Application" object class, the "Domain"

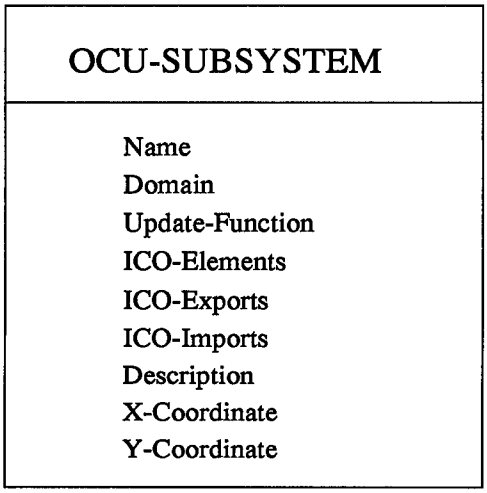


Figure 3.8 Object Model of an OCU Subsystem

attribute of the "OCU-Subsystem" object class identifies the domain from which the subsystem was composed. In a multiple domain application, the "Domain" attribute refers to

the primary domain from which the subsystem was composed; the use of alternate domains is not indicated by this attribute.

The "ICO-Elements" attribute means "is composed of elements." This attribute also makes use of aggregation since its type declaration is a set of "OCU-Element" objects. As a result, there are two levels of aggregation in an "OCU-Application" object—"OCU-Element" objects are nested within "OCU-Subsystem" objects, and "OCU-Subsystem" objects are nested within "OCU-Application" objects. Consistent with the OCU software architecture, an element can be either a subsystem or a primitive. Next, an analysis of the "OCU-Primitive" object is needed to understand how an "OCU-Application" object can contain primitive objects from more than one domain.

3.6.3 "OCU-Primitive" Object Class. A primitive is also modeled as an object class in the database. The object class is named "OCU-Primitive" and is shown with its attributes in Figure 3.9. The "Domain" attribute of an "OCU-Primitive" object refers to

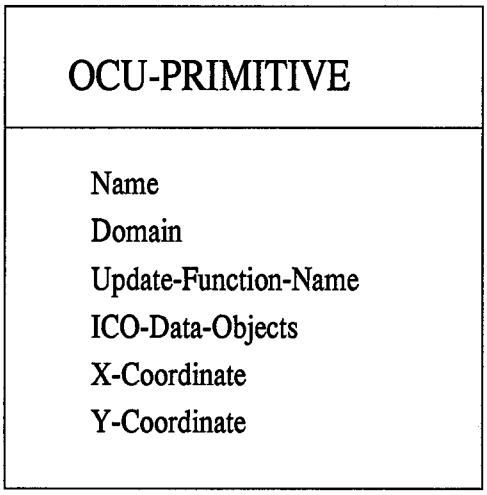


Figure 3.9 Object Model of an OCU Primitive

the domain in which the primitive belongs. In a multiple domain application, the domain of a primitive may differ from the domain of the subsystem and application to which it belongs. This difference in domains does not pose a problem in the database because the attribute "ICO-Elements" of an "OCU-Subsystem" object does not require all primitive objects nested within a given subsystem to be of the same domain.

3.6.4 Summary of Domain Sharing Techniques. The object-oriented database concept of aggregation was the primary technique identified in this analysis to allow the sharing of domain components. ITASCA allows aggregation in the schema definition by declaring the type of any object's attributes to be an object instance or a set of object instances. Aggregation allows an Architect application to be stored in the database as one object. One or more subsystems can be nested within each application. Also, one or more primitives and one or more subsystems can be nested within each subsystem. The primitives included in a subsystem do not have to be of the same domain as the subsystem and application. As a result, a single instance of an Architect application in the database can contain primitives from multiple domains.

The transformation functions discussed in Section 3.3.3 must keep track of the various domains involved in a multiple domain application. When loading an application from ITASCA to REFINE, the transformation functions use the "Domain" attribute of an "OCU-Primitive" object to determine if each primitive object is from an alternate domain. If so, the transformation functions must account for this in order to properly create the primitive in the REFINE object base. When saving an application to the database, the

transformation functions determine the domain of each primitive in the application so it can store the primitive in the proper object class within its domain.

3.7 Summary

This chapter began with an assessment of Architect's baseline operational capabilities and a discussion of the desired operational capabilities required to support multiple domain applications. Next, Architect's software environment was analyzed to explain the single domain limitation for applications in the baseline system. Since the software environment plays a different role in the original, file-based version of Architect than it does in the database version of Architect, those differences were analyzed. A discussion of Architect's semantic checks provided an understanding of how domains must be compatible to participate in a multiple domain application. The meta-model for domain definitions was analyzed due to the need to incorporate additional domains in the baseline database version of Architect. The chapter concluded with a discussion of object-oriented database techniques for sharing components across domain boundaries. In particular, aggregation allows applications to be composed of subsystems, and subsystems to be composed of primitives. As a final key point, the OODBMS does not require all primitives contained in a subsystem to belong to the same domain.

IV. Design and Implementation

4.1 Overview

This chapter describes the design and implementation of a multiple domain capability for Architect applications. The enhancements needed to support the use of primary and alternate domains while creating or editing an application are explained. At least one additional domain needed to be implemented in the database. This chapter explains the rationale for selecting the digital signal processing domain to complement the logic circuits domain. To cross domain boundaries, at least one pair of the logic circuits and digital signal processing primitives needed to be capable of exchanging (importing or exporting) data. This chapter describes the design and implementation of several new primitives in both domains, enabling the exchange of data across the domain boundaries. With these new primitives, the Architect system is capable of generating an application containing primitives from both domains.

4.2 Architect System Enhancements - Primary and Alternate Domains

In order for the Architect system to have the operational capabilities described in Option 2 of Section 3.2.2, several enhancements were implemented. These enhancements allowed for the use of primary and alternate domains during application composition.

The Architect design and implementation uses the concept of a *current* domain. Architect was originally designed such that it has one and only one current domain at any given point in time. Many of the activities required in Architect's application composition process depend on knowledge contained in the current domain. For example, when bring-

ing up the technology base of primitive objects, Architect produces a window containing primitives of the current domain. However, in a multiple domain environment, the current domain needs to change during the composition of a single application. At any given point in time, the primary domain or one of the alternate domains must be able to assume the role of current domain. As a result, enhancements were needed to manage the dynamics involved with the current domain for activities in the following key areas.

- Technology Base Window
- Object Editing
- Setting Icon Attributes
- Database Transformation Functions

A simple approach was taken to manage the domain dynamics. When performing an activity in one of the key areas on a primitive from an alternate domain, the alternate domain gets temporarily established as the current domain. When the activity is completed, the primary domain gets re-established as the current domain. This idea is illustrated in Figure 4.1. A discussion of the design and implementation for the required enhancements follows.

4.2.1 Technology Base Window. The mouse handler function for the technology base window allows the user to create instances of primitive objects from the technology base window and place them into the current subsystem window. Also, the mouse handler function provides the user a menu of several additional options by clicking on the diagram surface. The option to select primitives from an alternate domain was added to this

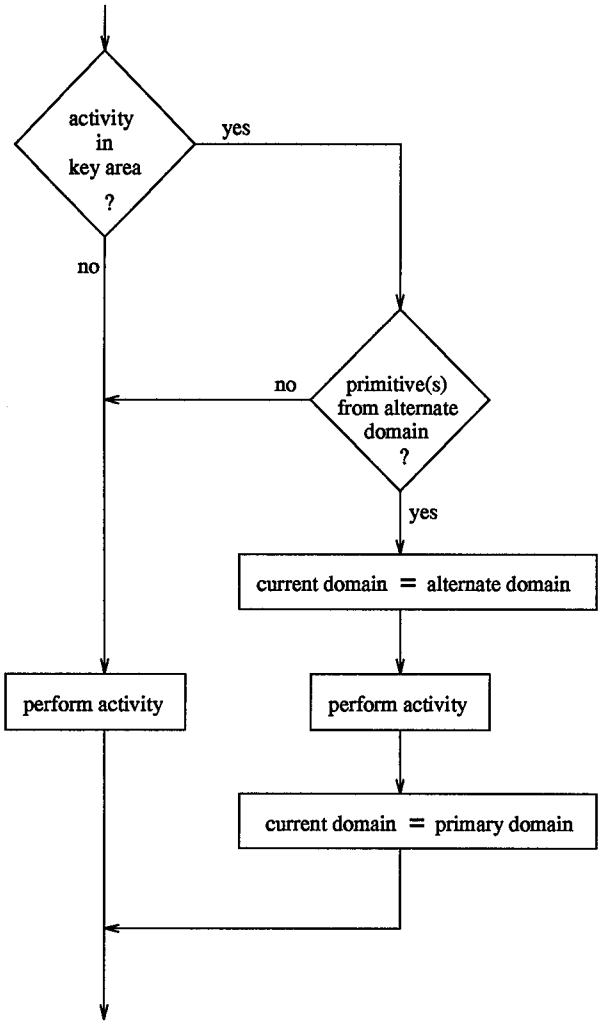


Figure 4.1 Management of Domain Dynamics

menu. When the user chooses this option, a function is invoked that asks the user to specify an alternate domain. If the alternate domain has not been previously loaded from the database into REFINE, it is loaded at this time. At this point, the mouse handler function establishes the alternate domain as the current domain in the Architect system, consistent with Figure 4.1. The mouse handler function then invokes another function to display the technology base window for the current domain. This allows the user to place instances of primitives from the alternate domain into the current subsystem window.

After all instances from the alternate domain have been created, the primary domain is re-established as the current domain.

4.2.2 Object Editing. Each primitive object in an Architect application has some number of editable attributes. The user may start the editing process by requesting to edit an application, or a subsystem of an application. In either case, a window displaying the components of the application or subsystem is displayed. The user then clicks on the object to be edited. Architect provides a menu of options, including the option to view and edit the object's attributes. If the user selects that option, a function to display the list of editable attributes for the object is invoked. Next, the user may select any attribute in the list and edit its value.

An enhancement was needed for the object editing process to allow objects from an alternate domain to be edited. When the function to display the list of editable attributes for an object is invoked, the function must ensure the object's domain is the current domain since the retrieval of attributes is limited to objects of the current domain. Thus, the enhancement here implements the management of domain dynamics of Figure 4.1.

4.2.3 Setting Icon Attributes. The visual interface for Architect builds many icons for primitive objects during the application composition process. For example, the technology base window displays an icon for each primitive in a given domain. Also, the windows that are generated during editing of an application or subsystem include an icon for each object instance they contain. To build the icons, the Architect Visual System Interface (AVSI) invokes a function to set the icon attributes for a primitive as specified

by a bitmap associated with the primitive. The attributes for an icon define the physical appearance of the icon when displayed.

The baseline Architect system is implemented such that the function to set the icon attributes can only be invoked for the primitives of the current domain. Thus, an enhancement was implemented similar to the enhancements described in the preceding sections, consistent with Figure 4.1.

4.2.4 Database Transformation Functions. The database transformation functions discussed in Section 3.3.3 required an enhancement to save and load multiple domain applications. A discussion of this implementation follows.

4.2.4.1 Saving Applications. When saving an application to the database, Architect's transformation functions create an instance of the "OCU-Primitive" object class in the ITASCA database for each instance of a REFINE primitive in the application. "Domain" is an attribute of an ITASCA "OCU-Primitive" instance and specifies the domain to which the primitive belongs. However, the design and implementation of the transformation functions require the value of the primitive's domain to be set to the current domain. Since the domain of the application being saved is the current domain, an enhancement was implemented to accurately build ITASCA primitives from an alternate domain. Once again, the enhancement adhered to the logic of Figure 4.1.

4.2.4.2 Loading Applications. This enhancement is similar to the one required for saving applications. When loading an application from the database, Architect's transformation functions create an instance of a primitive in the REFINE object base for

every ITASCA "OCU-Primitive" object instance contained in the application. However, the design of the transformation functions limits the creation of REFINE primitives to those of the current domain. Since the domain of the application being loaded is the current domain, an enhancement consistent with Figure 4.1 was implemented so primitives from an alternate domain can also be created.

4.3 Selection of Additional Domains

After enhancing the Architect system to allow the use of primary and alternate domains, the next step was to implement at least one new domain in the database. The selection of the new domain must accommodate the fact that the domains participating in a multiple domain application must have primitive objects capable of interfacing with each other as explained in Section 3.4. This presented two alternatives.

1. Implement at least one additional domain capable of interfacing with the logic circuits domain.
2. Implement at least two additional domains capable of interfacing with each other.

The first alternative was selected. Actually, two additional domains were implemented. The first was really an extension of the original logic circuits domain, but was implemented as a separate domain. This domain was named "Circuits-Additional" and added additional logic circuits primitives. This served two practical purposes. First, useful logic circuits primitives were added to the Architect technology base. Second, the "Circuits-Additional" domain is completely compatible with the original logic circuits domain. This

provided a simple and ideal environment for building multiple domain applications, aiding the development and validation of the enhanced Architect system.

The digital signal processing (DSP) domain was the second domain implemented. This domain was implemented in the file-based version of Architect during previous research by Warner (23). The possibility of DSP primitives importing or exporting binary signals made this an attractive domain to combine with logic circuits in a multiple domain environment.

4.4 Implement "Circuits-Additional" Domain

To implement the "Circuits-Additional" domain, several tasks were accomplished. Specific primitives were selected for incorporation into the domain. The update functions for each primitive were designed and written. The database schema for the domain was designed and implemented. Information pertaining to the domain definition meta-model, which was analyzed in Section 3.5, was developed and entered into the ITASCA database. This enabled the REFINE source code for the domain to be generated. This section addresses these tasks.

4.4.1 Selection of Additional Primitives. As stated earlier, the objective of developing the "Circuits-Additional" domain was to add useful logic circuits primitives to the Architect technology base and to provide a simple environment to support multiple domain applications. Therefore, the following small set of logic circuits primitives was selected for implementation in the new domain.

- And-Gate-3Input - This primitive imports three binary signals, performs a "logical and" of the three signals, and outputs the result.
- Or-Gate-3Input - This primitive imports three binary signals, performs a "logical or" of the three signals, and outputs the result.
- Full-Adder - This primitive imports three binary signals—an addend, an augend, and a carry-in. The output is two binary signals—the sum and carry-out.
- Full-Adder-4Bit - This primitive imports two binary numbers, each number containing four bits. The output is the four-bit sum and a carry-out bit.
- Full-Subtractor - This primitive imports three binary signals—a minuend, a subtrahend, and a previous borrow. The output is two binary signals—the difference and an output borrow.
- Full-Subtractor-4Bit - This primitive subtracts one binary number from another, each number containing four bits. The output is the four-bit difference and a borrow bit.

The significance of adding the adder and subtractor primitives will become more apparent later since they participate with digital signal processing primitives to form a multiple domain application.

The "Switch" and "LED" (light emitting diode) primitives, which are part of the original logic circuits domain, were also included in this domain. These primitives are necessary to build meaningful applications within the "Circuits-Additional" domain. The design and implementation of these primitives were simply reused from the original logic circuits domain.

Table 4.1 Truth Table for 3-Input And-Gate

Imports			Exports
In1	In2	In3	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 4.2 Truth Table for 3-Input Or-Gate

Imports			Exports
In1	In2	In3	Out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

4.4.2 Update Functions for the New Primitives. Domain information to develop the new logic circuits primitives was obtained from Mano's book, *Digital Logic and Computer Design* (15). Truth tables for the "And-Gate-3Input," "Or-Gate-3Input," "Full-Adder," and "Full-Subtractor" are illustrated in Tables 4.1, 4.2, 4.3, and 4.4, respectively. The size of the truth tables for the four-bit full-adder and four-bit full-subtractor makes them impractical to present. The truth tables identify the exports generated by each primitive's update function for all possible combinations of imports. The REFINE update functions for each of these primitives is located in Appendix B.

4.4.3 Object Model for Schema Implementation. Before implementing the schema for the "Circuits-Additional" domain, an object model for the domain needed to be de-

Table 4.3 Truth Table for Full-Adder

Imports			Exports	
Addend	Augend	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 4.4 Truth Table for Full-Subtractor

Imports			Exports	
Minuend	Subtrahend	Previous Borrow	Difference	Output Borrow
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

veloped. The object model diagram is shown in Figure 4.2. This model was developed in

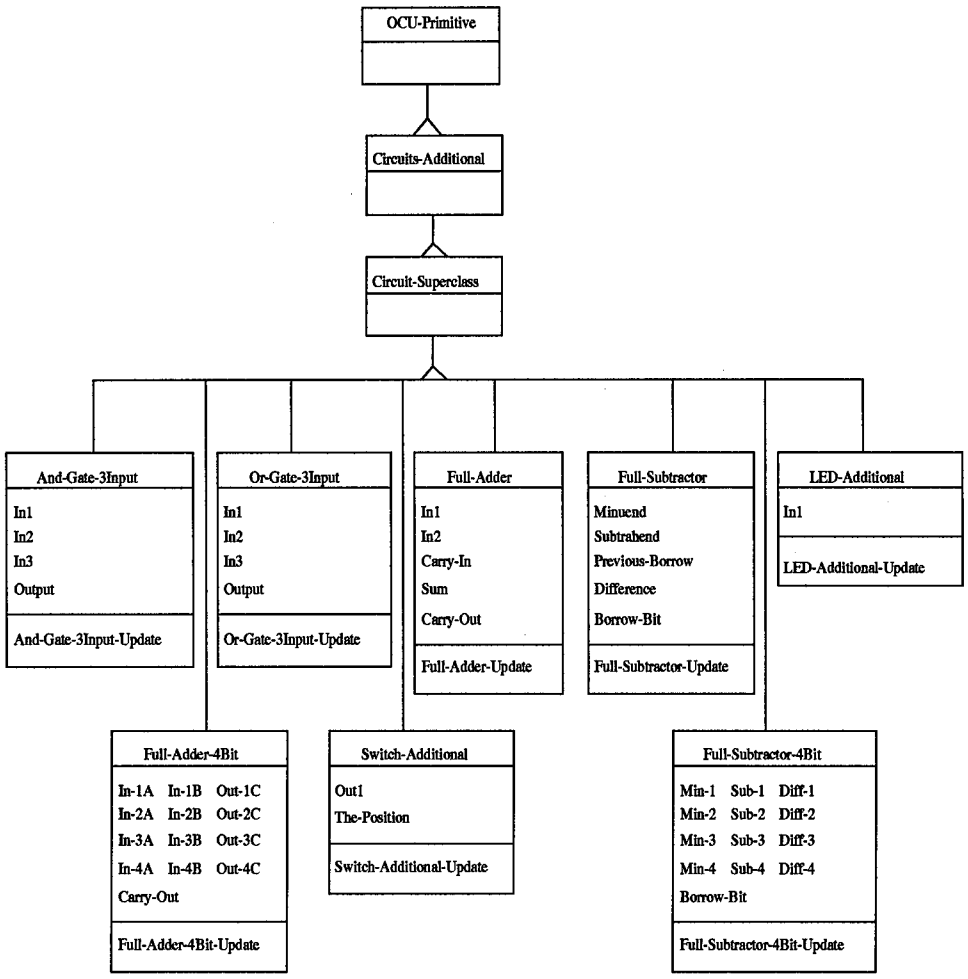


Figure 4.2 Object Model for "Circuits Additional" Domain

accordance with the meta-model for domain definitions.

"Circuits-Additional" is the top level object class and corresponds to the domain definition class of the meta-model. This class is simply a container class for the domain and can have no attributes. The design of Architect and the database requires each domain to inherit from the OCU software architecture. Therefore, "Circuits-Additional" is a subclass of the "OCU-Primitive" class in the database schema. Note, the "OCU-Primitive" class is

not part of the "Circuits-Additional" domain. It is actually part of the OCU architecture and is shown here to illustrate the connection between the domain and software architecture models. All the concrete and abstract classes in the domain are subclasses of the "Circuits-Additional" class.

Each primitive in the domain corresponds to a unique concrete class and is a leaf class in the diagram. Since the primitives are concrete classes, each has a REFINE update function. The diagram identifies each primitive with its update function. The update functions are not ITASCA methods. They are simply the REFINE update functions required by Architect for its primitives. The update functions are stored as REFINE source code in the ITASCA database. Note that the switch primitive is named "Switch-Additional" and the LED primitive is named "LED-Additional." This avoids a naming conflict with the "Switch" and "LED" classes in the schema of the original logic circuits domain.

The object model contains one abstract class named "Circuit-Superclass." This class really serves no purpose with the given design since there were no common attributes in the domain that could be inherited by all primitives. However, this abstract class was included as a subclass of "Circuits-Additional" in the schema because it might provide opportunities for inheritance in future designs.

The attributes in the object model diagram belong to one of the following categories: OCU-ATTRIBUTE, OCU-INPUT, or OCU-OUTPUT. All attributes belonging to the OCU-ATTRIBUTE category are included as attributes in the schema. The attributes belonging to the OCU-INPUT and OCU-OUTPUT categories are not included as attributes in the schema. Instead, they represent the imports and exports for the primitive classes of

the domain. These attributes are necessary to accurately generate the REFINE source and executable code for the domain. The next section summarizes the domain definition data used to generate the REFINE code.

4.4.4 Domain Definition Inputs. Domain definition data was input into the database as specified by the object model from the previous section. The data was input into the following five ITASCA object classes.

- Data Object - There were 47 instances of data objects. The data objects correspond to the attributes illustrated in the various object classes of Figure 4.2.
- Concrete Object - There were eight instances of concrete classes corresponding to the primitives of the domain. These are illustrated in Figure 4.2.
- Abstract Object - There was one instance of an abstract class, "Circuit-Superclass."
- REFINE Function - There were eight instances of REFINE functions, one for each concrete class.
- Domain Definition - There was one instance of a domain definition.

After the domain definition data was loaded into the ITASCA database, the "make-refine-file" method of the "Domain Definition" class was executed. This generated the REFINE source code for the domain. Ultimately, the source code was compiled and the resultant executable code was stored in the database. With the database version, all of Architect's executable files are stored in the database and are loaded into REFINE as needed during an Architect session.

Table 4.5 DSP Primitives

Stored-Signal	Sinusoid	Noise	Time-Filter
Piecewise-Linear	Print-Signal	Save-Signal	Reverse-Signal
Unit-Sample-Sequence	Input-Buffer	Delay	Window-Signal
Unit-Step-Sequence	Output-Buffer	Adder	Truncate-Signal
Graph-1-Signal	Multiplier	Signal-Adder	User-Designed-Filter
Graph-2-Signal	Signal-Multiplier	DFT	Complex-To-Real
Graph-3-Signal	Signal-Subtractor	IDFT	Scale-Signal
Graph-4-Signal	Signal-Divider	Convolution	Real-To-Complex
Signal-Abs-Dif	Frequency-Filter	Pad-Signal	

4.4.5 Icon Bitmap Construction and Implementation. AVSI requires each primitive to be associated with an icon to allow the primitive objects to be displayed during an Architect session. The bitmaps for the "Circuits-Additional" icons were built using Icon Editor, an OPENWINDOWS™ tool. The data for each bitmap was then reformatted as a list to allow its storage in the "Icon-Obj" object class of the database. The COMMON WINDOWS function "bitmap-to-expr" can be used to convert bitmap data into a list. With this implementation, AVSI can load its icons from the database rather than files. After loading a list containing icon data, AVSI calls the COMMON WINDOWS function "expr-to-bitmap" to return a bitmap whose image is created from the list (9:4-12). For a thorough discussion of icon design and development for the Architect system, refer to Appendix B of Cossentine's thesis (8).

4.5 Implement Digital Signal Processing Domain

The DSP domain contains 35 primitives as implemented by Warner in the file-based version of Architect. All of these primitives are identified in Table 4.5. Warner's design and implementation of these primitives was reused for entry into the database version of

Architect. For a complete discussion of Architect's DSP domain, consult Warner's thesis (23).

To implement the DSP domain, many of the same tasks required to implement the "Circuits-Additional" domain were accomplished. The database schema was designed and implemented. Information pertaining to the domain definition meta-model was developed and entered into the ITASCA database. This allowed the REFINE source and executable code for the domain to be generated. This section addresses these tasks.

4.5.1 Object Model for Schema Implementation. Similar to the development of the "Circuits-Additional" domain, an object model for the DSP domain was needed. The model, developed in accordance with the domain definition meta-model, is shown in Figure 4.3. "DSP" is the top level object class and corresponds to the domain definition class of the meta-model. "DSP" has ten abstract subclasses. Nine of the ten abstract classes have concrete subclasses. There are a total of 35 concrete classes, one for each primitive in the domain. The concrete classes are grouped into single boxes according to their superclass. Grouping more than one class in a box deviates from Rumbaugh's Object Modeling Technique (OMT). This approach is more space efficient in this case and allows all primitive classes in the domain to be displayed reasonably in one diagram. However, the attributes for all the object classes do not reasonably fit into the diagram and are omitted.

Similar to the "Circuits-Additional" domain, the ITASCA schema for the DSP domain was built according to the object model. For a complete representation of the DSP object

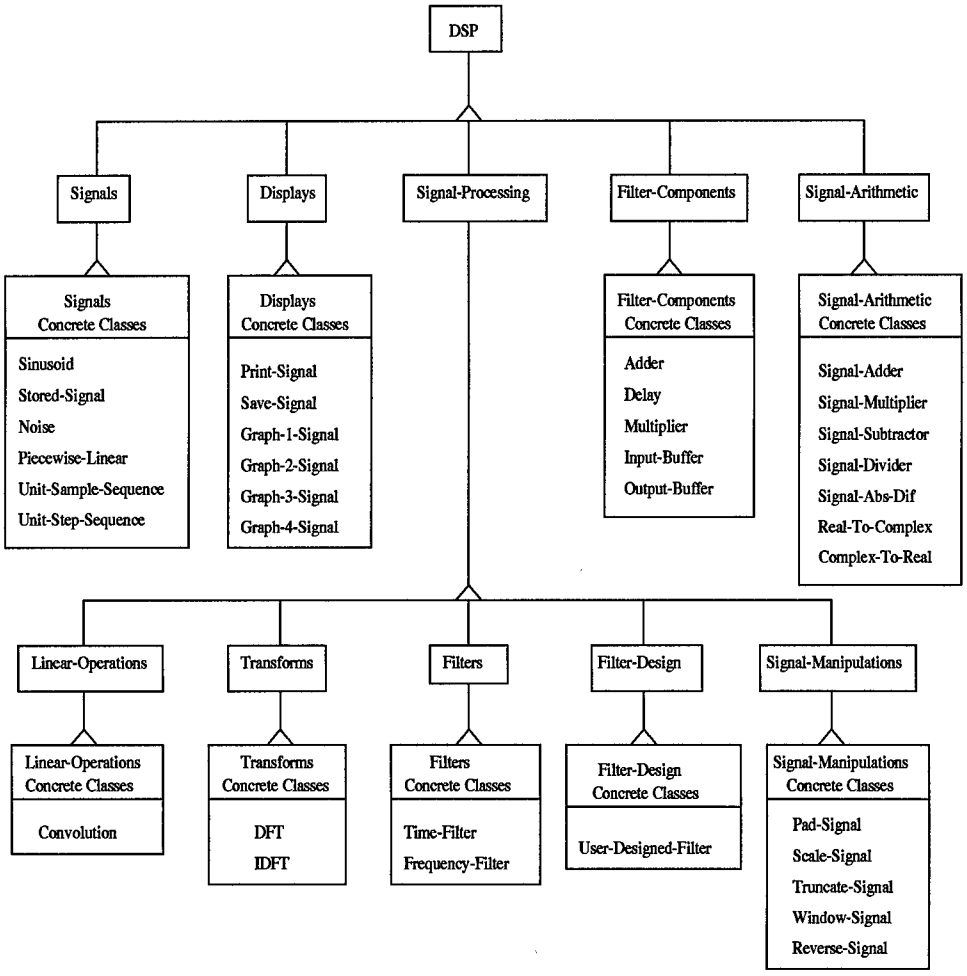


Figure 4.3 Object Model for DSP Domain

model, refer to the diagrams in Appendix C. These diagrams adhere to Rumbaugh's OMT and show all attributes for each class.

4.5.2 Domain Definition Inputs. The following summarizes the domain definition data that was entered into the database for the DSP domain.

- Data Object - There were 119 instances of data objects. The data objects correspond to the attributes contained in the various diagrams in Appendix C.

- Concrete Object - There were 35 instances of concrete classes corresponding to the primitives of the domain.
- Abstract Object - There were ten instances of an abstract class.
- REFINE Function - There were 35 instances of REFINE functions, one for each concrete class.
- Domain Definition - There was one instance of a domain definition.

As with the "Circuits-Additional" domain, after the domain definition inputs were completed for the DSP domain, the REFINE code was generated and stored in the database.

4.6 Digital Signal Processing Domain Enhancements

In spite of implementing the "Circuits-Additional" and DSP domains, the ability to compose multiple domain applications was limited. The "Circuits-Additional" primitives were fully compatible with the original logic circuits primitives. Therefore, applications containing primitives from both logic circuits domains were possible. While this provided a good basis for doing some initial testing and validation of Architect's multiple domain enhancements, it did not provide any meaningful applications. This is because the primitives of the two logic circuits domains would not be implemented as separate domains in a realistic environment. Instead, they would have been merged into one domain.

To generate more meaningful applications, the compatibility of the DSP and logic circuits domains was assessed. For primitives to exchange data, they must comply with Architect's semantic checks explained in Section 3.4. All of the logic circuits primitives have import or export areas with a type of "binary" and category of "signal." None of the

DSP primitives were compatible with any of the logic circuits primitives. However, the possibility of augmenting the DSP domain with compatible primitives was promising.

Two candidates for addition to the DSP domain were an analog-to-digital converter and a digital-to-analog converter. In digital signal processing, an analog-to-digital converter generates a binary code. This binary code could be exported to primitives in the logic circuits domain for manipulation. The logic circuits primitives could then export the manipulated binary code to a digital-to-analog converter, which receives a binary code as input and outputs its analog equivalent.

4.6.1 Design of an Analog-to-Digital Converter. An analog-to-digital converter (ADC) samples an analog signal at some sampling rate or time interval. It converts each sample into a representative binary code. For example, a signal might represent the voltage for some electrical circuit. The ADC assesses the value of the voltage of the input sample and assigns the sample a binary code representing a quantized value or level. The ADC contains a number of quantization levels. The assigned binary code corresponds to the quantized value that most closely approximates the amplitude of the input (16:114).

The full scale level of the ADC is the maximum amplitude it supports. To allow for negative values, the full range of values supported by the ADC is two times the full scale level. This range of values is divided into a fixed number of discrete quantization levels. As a result, the step size for each quantization level, denoted as Δ, is equal to two times the full scale level divided by the number of quantization levels (16:117-118).

Since each sample from the analog signal is assigned to the nearest quantization level, the analog-to-digital conversion is not completely precise. The error generated by digital

conversion of a sample is less than or equal to one half of the step size for the quantization levels. Therefore, the precision of an ADC can be increased by merely increasing the number of quantization levels, which decreases the step size. Samples above the most positive quantization level are assigned to the highest level; those below the most negative quantization level are assigned to the lowest level.

The number of quantization levels is limited by the number of bits used in the binary code. This relationship is defined by the formula, $y = 2^n$, where y is the number of quantization levels and n is the number of bits in the binary code. Since the number of quantization levels increases exponentially with the number of bits, increasing the size of the binary code decreases the step size and improves the precision of the analog-to-digital conversion. A 4-bit code was used for the ADC implemented for Architect, resulting in a maximum of 16 quantization levels.

Each quantization level corresponds to a unique binary code. Several binary coding schemes exist and a couple of them are shown for a 4-bit ADC in Table 4.6. The offset binary code assigns a numeric ordering of the quantization levels, beginning with the most negative level. With a 4-bit ADC, the levels are assigned offset binary codes representing the values 0 through 15. However, in digital signal processing, it is sometimes preferable to do arithmetic directly where the code is a scaled representation of the quantized samples. The two's complement code provides this capability. Two's complement is obtained by complementing the most significant bit of the offset binary code. This is a convenient system of representing signed numbers and is used in most computers. Each two's complement code is associated with a specific quantized level, ranging from -8Δ to 7Δ (16:116-117).

Table 4.6 Binary Codes for 4-Bit Analog-to-Digital Converter

Offset Binary	Two's Complement	
Code	Code	Quantized Level
1111	0111	7Δ
1110	0110	6Δ
1101	0101	5Δ
1100	0100	4Δ
1011	0011	3Δ
1010	0010	2Δ
1001	0001	1Δ
1000	0000	0Δ
0111	1111	-1Δ
0110	1110	-2Δ
0101	1101	-3Δ
0100	1100	-4Δ
0011	1011	-5Δ
0010	1010	-6Δ
0001	1001	-7Δ
0000	1000	-8Δ

The following example shows how addition can be performed directly on two's complement codes. The codes 0110 and 1011 correspond to the quantized levels 6Δ and -5Δ, respectively. The binary sum of 0110 and 1011 is 0001, disregarding the most significant carry bit. The two's complement code 0001 corresponds to the quantized level of 1Δ, which is expected when summing 6Δ and -5Δ. A similar addition of the corresponding offset binary codes does not provide useful results.

For another example, consider the following subtraction. Two's complement codes of 1110 and 1100 correspond to the quantized levels of -2Δ and -4Δ, respectively. Subtracting 1100 from 1110 yields 0010. The two's complement code 0010 corresponds to the quantized level of 2Δ, which is expected when subtracting -4Δ from -2Δ. Again, a similar subtraction of the corresponding offset binary codes does not provide useful results.

Next, refer to Figure 4.4 for an example of how samples from an analog signal are converted to a binary code. The first sample is taken at time t_0. Since the amplitude of this

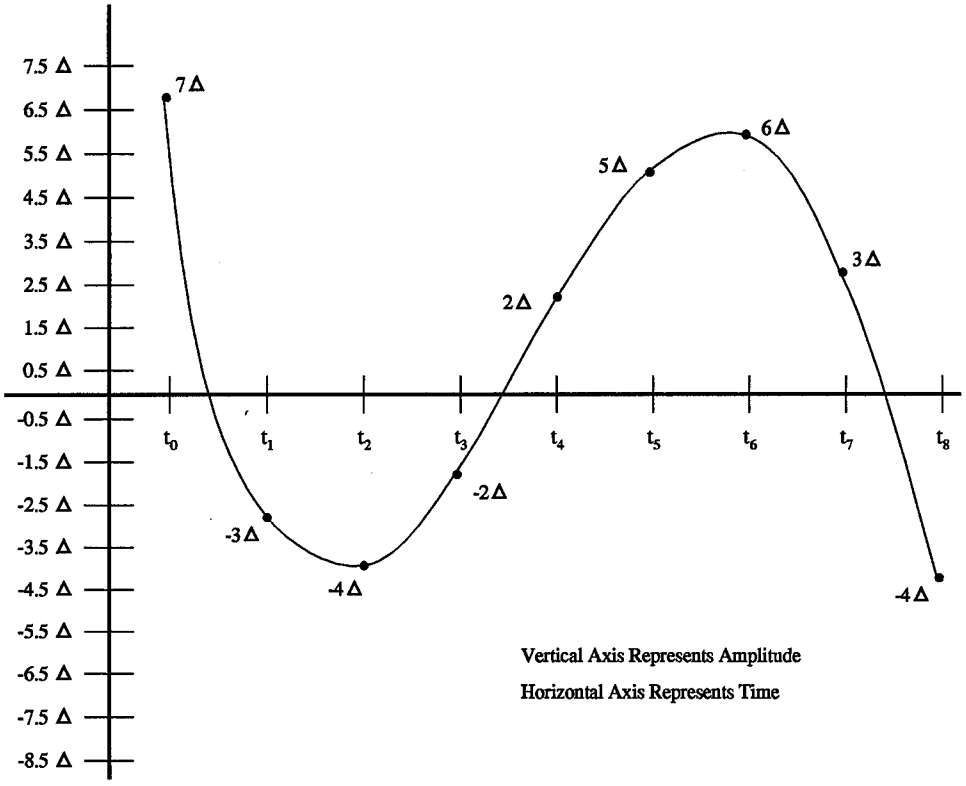


Figure 4.4 Sampling of an Analog Signal

sample is between 6.5Δ and 7.5Δ, it is assigned the quantized value of 7Δ and the two's complement code of 0111. The second sample is taken at time t_1. Since the amplitude of this sample is between -2.5Δ and -3.5Δ, it is assigned the quantized value of -3Δ and the two's complement code of 1101. Each of the remaining samples are processed in the same fashion.

4.6.2 Design of an Digital-to-Analog Converter. A digital-to-analog converter (DAC) receives a binary code representing a sample of an analog signal and converts the binary code to its analog equivalent. As implemented in Architect, the DAC primitive basically reverses the actions performed by the ADC. Therefore, the discussion here is brief. When the DAC receives a binary code, it determines which quantized level the

code represents. The value of the quantized level is an estimate of the amplitude for that particular sample. When all samples have been converted from binary to analog form, a curve can be fit through the samples providing an approximate representation of the analog signal. As with the analog-to-digital converter, the accuracy of the digital to analog converter increases as the size of the binary code increases.

4.6.3 Implementation Actions. This section briefly summarizes the actions required to implement the design of the DAC and ADC primitive classes. An object model including the new classes was developed. The portion of the diagram containing the new classes is shown in Figure 4.5. Update functions for both primitives were written and

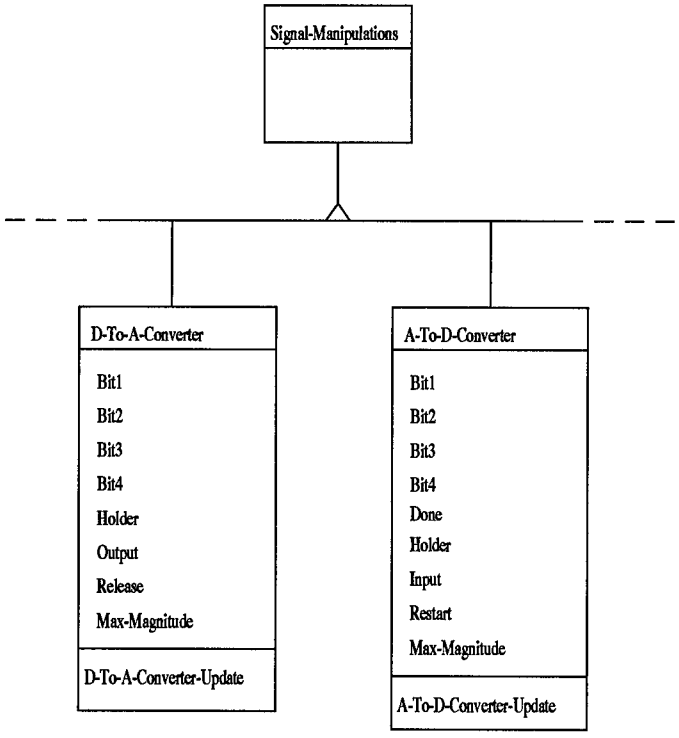


Figure 4.5 Object Model for ADC and DAC Primitives

are located in Appendix B. The database schema was modified to include the two new

DSP classes. They were both modeled as subclasses of the "Signal-Manipulations" class. To enable enhancement of the REFINE source and executable code for the DSP domain, additional domain definition data was entered into the database for the ADC and DAC primitives. The data is summarized as follows.

- Data Object - There were 17 instances of data objects. The data objects correspond to the attributes contained in the diagram of Figure 4.5.
- Concrete Object - There were 2 instances of concrete classes.
- REFINE Function - There were 2 instances of REFINE functions, one for each primitive class.

The final action taken was developing icon bitmaps for the new primitives. Icon Editor was used for that purpose.

4.7 Summary

This chapter began with a description of the design and implementation of a multiple domain capability for Architect applications. The Architect system was enhanced to support the use of primary and alternate domains while creating or editing an application. Next, the need for implementing at least one new domain in the database was discussed. Since the logic circuits domain was already implemented in the baseline database version of Architect, the digital signal processing domain was implemented due to its potential compatibility with the logic circuits domain. Finally, this chapter discussed the design and implementation of several new primitives in both domains, making the two domains

compatible. These new primitives make it possible for the two domains to exchange binary signals within a single application.

V. Testing and Validation

5.1 Overview

This chapter explains the testing and validation of Architect's multiple domain application capability. Testing was actually performed incrementally during the development process. As mentioned in Section 4.3, the implementation of the "Circuits-Additional" domain, in conjunction with the original logic circuits domain, provided a simple environment for doing initial testing and validation. Later, more substantial testing was possible after implementing Warner's DSP domain along with the analog-to-digital and digital-to-analog converters.

5.2 Objectives

Testing and validation of this research effort needed to confirm a set of primary and secondary objectives as follows.

- Primary Objectives
  - Display the technology base window for alternate domains when appropriate.
  - Display icons for primitives from alternate domains in multiple domain applications.
  - Edit objects from alternate domains in multiple domain applications.
  - Successfully save multiple domain applications from the REFINE object base to the database.

  - Successfully load multiple domain applications from the database into the REFINE object base.
  - Properly execute multiple domain applications.
- Secondary Objectives
  - Ensure the accuracy of the new logic circuits domain primitives.
  - Ensure the accuracy of the new DSP domain primitives.

The primary objectives pertain to the design and implementation of the Architect system enhancements described in Section 4.2, which allow for the use of primary and alternate domains. The secondary objectives pertain to the accuracy of the new domain knowledge implemented in the technology base. While the knowledge in the original logic circuits and DSP domains was validated by previous research, the new primitives such as the full-adder and the digital-to-analog converter needed validation.

The objectives pertaining to the Architect system enhancements are considered primary because they relate directly to the problem statement of this thesis. These objectives provide the infrastructure enabling the composition of multiple domain applications for *any* compatible domains. The secondary objectives were needed to ensure the testing and validation of the primary objectives were not impacted by inconsistencies of the new primitives.

5.3 Testing of Primary Objectives

After the objectives for testing and validation had been identified, a series of tests were conducted. A discussion of the tests and results follows.

5.3.1 Technology Base Window for Alternate Domains. Displaying the technology base window for alternate domains was simple to confirm. Testing demonstrated the window for any of the available alternate domains could be displayed during the application composition process. However, an equally important consideration was confirming the primary domain gets re-established as the current domain after the technology base window for an alternate domain is deactivated. Recall from Chapter IV, when an activity is completed that requires an alternate domain to be established as the current domain, the primary domain should be re-established as the current domain. Testing confirmed that a request to redisplay the technology base window after previously displaying an alternate domain would result in the reappearance of the primary domain. This was a crucial result demonstrating the dynamics of the current domain are properly managed.

5.3.2 Display Icons for Alternate Domains. After including primitives from an alternate domain in an application, testing needed to confirm icon images from both the primary and alternate domains could be displayed in a single window. This is required for several windows generated during application composition. Testing confirmed the satisfaction of this objective. An example of a subsystem window is located in Figure 5.1. This window contains a sinusoid primitive and an analog-to-digital converter primitive from the DSP domain. It also contains four LED primitives from the logic circuits domain. For this application, the DSP domain was the primary domain. The imports/exports window for this application is shown in Figure 5.2.

5.3.3 Edit Objects from an Alternate Domain. After displaying primitives from an alternate domain, testing needed to confirm those primitives could be edited. This

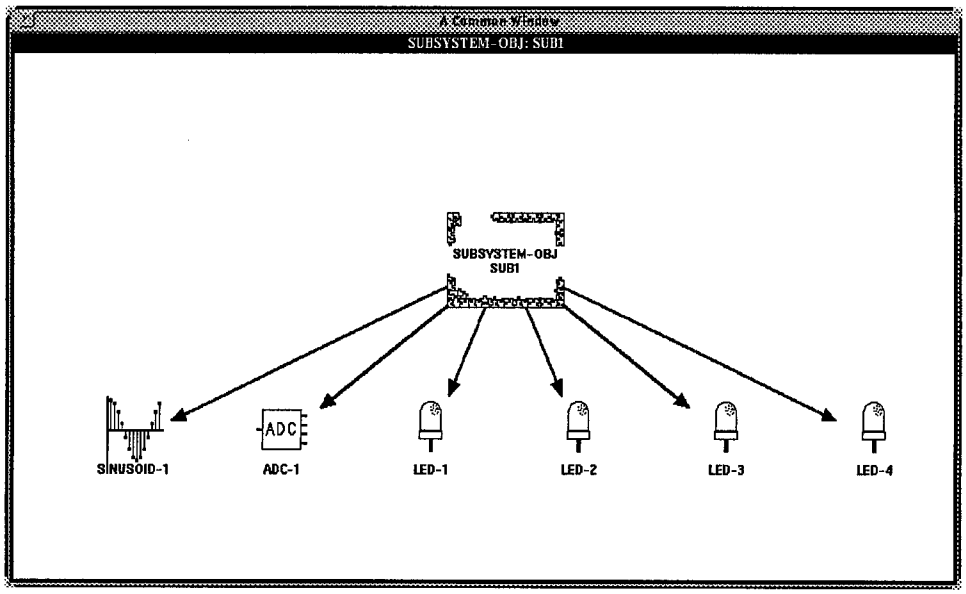


Figure 5.1 Subsystem Window

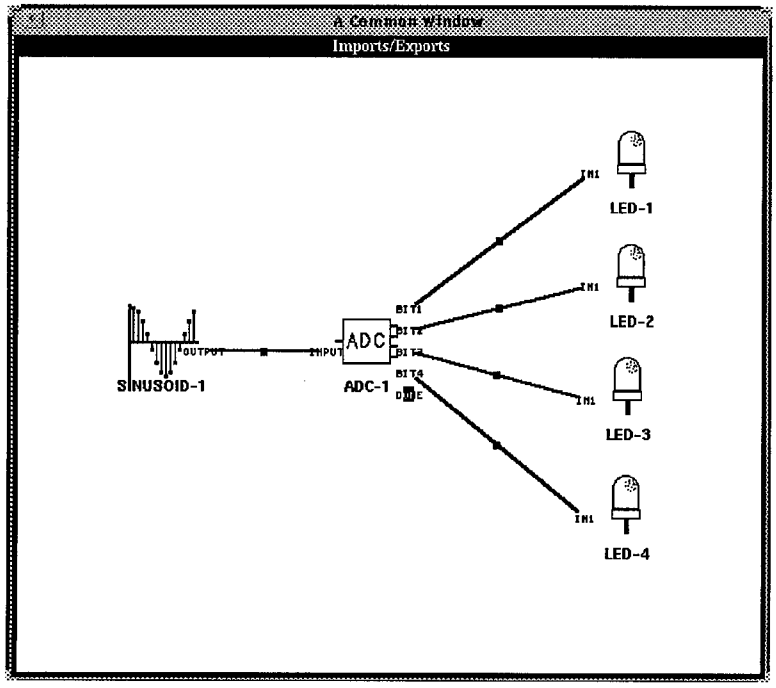


Figure 5.2 Imports/Exports Window

objective relates to the Architect enhancement that allows the display of editable attributes for a primitive from an alternate domain. Figure 5.3 shows a list of editable attributes generated during testing for one of the LED primitives of Figure 5.1, confirming the list of

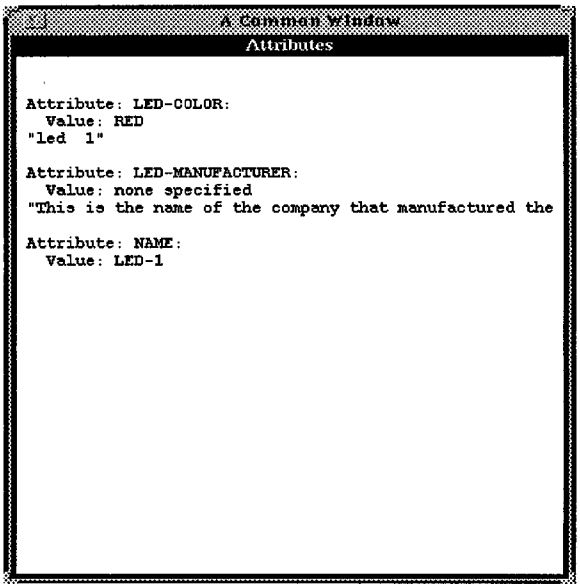


Figure 5.3 Editable Attributes

editable attributes gets properly displayed. Testing also confirmed objects can be edited by clicking on the desired attribute of Figure 5.3 and changing its value.

5.3.4 Save Multiple Domain Applications. After composing a multiple domain application in the REFINE object base, testing needed to confirm the application could be saved to the database. This objective relates to the enhancement of the database transformation functions that allows any subsystem of a given application to contain primitives from an alternate domain.

Testing confirmed the satisfaction of this objective. Inspection of multiple domain applications saved in the "OCU-Application" object class of the database revealed that

all of the primitive objects contained in a subsystem get properly aggregated into the subsystem. This includes primitives from an alternate domain. Recall from Section 3.6, in the database, subsystems are nested within applications, and primitives are nested within subsystems. Testing demonstrated that each "OCU-Primitive" instance gets stored in its proper object class of the database schema.

5.3.5 Load Multiple Domain Applications. After saving a multiple domain application in the ITASCA database, Architect must be able to load it back into the REFINE object base. As with the objective of the preceding section, this objective relates to the enhancement of the database transformation functions. Testing confirmed that when loading a multiple domain application from the database, all primitives get properly restored in the REFINE object base, including those from an alternate domain.

5.3.6 Execute Multiple Domain Applications. Architect must be able to execute multiple domain applications after they have been composed. Architect must also be able to execute previously saved multiple domain applications after they have been loaded from the database. Successful execution of an application requires each instance of a primitive contained in the application to be associated with the correct REFINE update function. Testing demonstrated that this association is maintained correctly in the multiple domain environment. As a result, all multiple domain applications, including those loaded from the database, executed properly during testing.

5.4 Testing of Secondary Objectives

Validation of the secondary objectives depended primarily on the accuracy of the REFINE update functions for the new primitive classes implemented in the logic circuits and DSP domains. The new logic circuits domain primitives were tested by assessing their consistency with the truth tables from Chapter IV. All possible combinations of inputs were tested to ensure the correct outputs get generated in all cases. Testing results demonstrated each of the logic circuits primitives was implemented correctly.

The new DSP domain primitives were tested by composing an application that supplies a sinusoid to an analog-to-digital converter, which outputs four binary signals (a four-bit code) for each sample of the sinusoid. The binary signals are delivered to a digital-to-analog converter, which outputs an approximation of the original sinusoid. Both the original sinusoid and the output of the digital-to-analog converter are supplied as input to a graph for comparison. Figure 5.4 is the Architect Imports/Exports window which shows the configuration for this application. Notice that a fifth binary signal is generated by the analog-to-digital converter and delivered to the digital-to-analog converter. This signal is set equal to true when the analog-to-digital converter has processed the last sample of the sinusoid. At this point, the digital-to-analog converter becomes aware that it has a complete signal ready for output to the graph.

During execution, Architect applications containing a graph primitive spawn a Khoros graph, which is displayed on the workstation screen. The graph in Figure 5.5 displays the two sinusoids. One is the sinusoid supplied directly to the graph, indicated by the solid plot. The other is the sinusoid output by the digital-to-analog converter, indicated by the

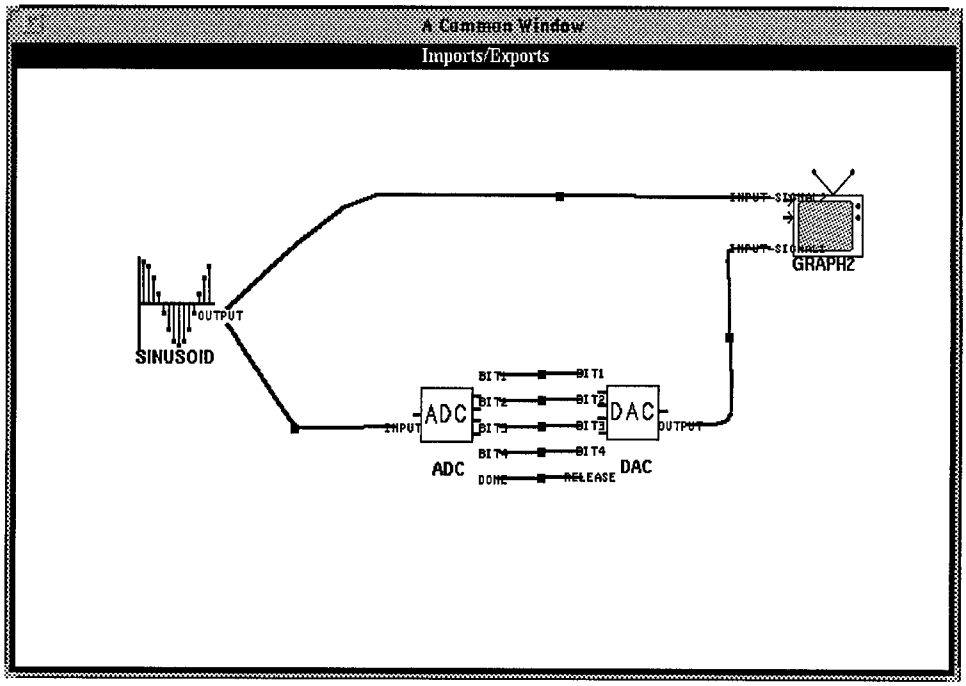


Figure 5.4 ADC-DAC Example

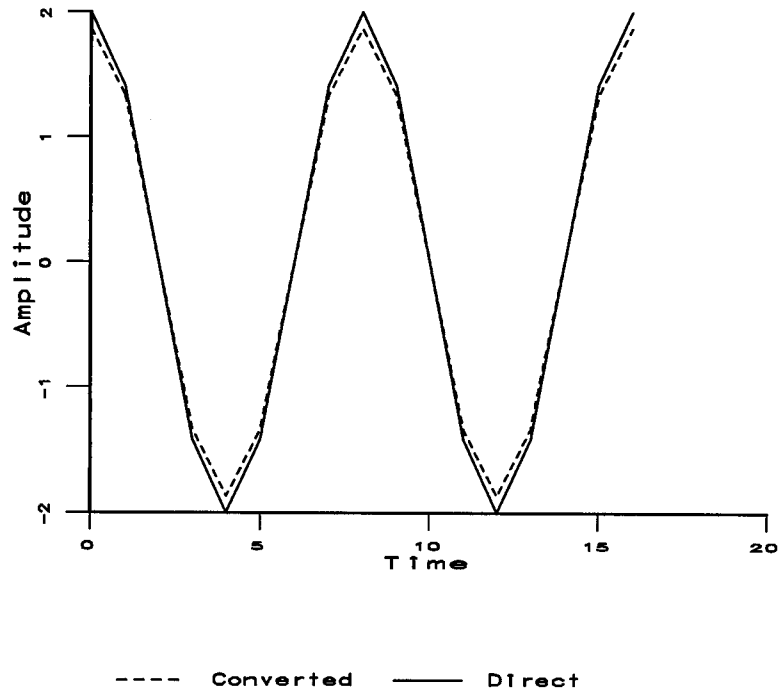


Figure 5.5 Sinusoid Plots

dashed plot. The two signals almost overlay each other. The fact that they do not precisely overlay each other indicates a small amount of error was introduced by the analog-to-digital and digital-to-analog conversions. This, of course, was expected.

5.5 Consolidated Example

At this point, an example of a multiple domain application is presented that neatly consolidates the testing and validation objectives. Figure 5.6 shows the Imports/Exports

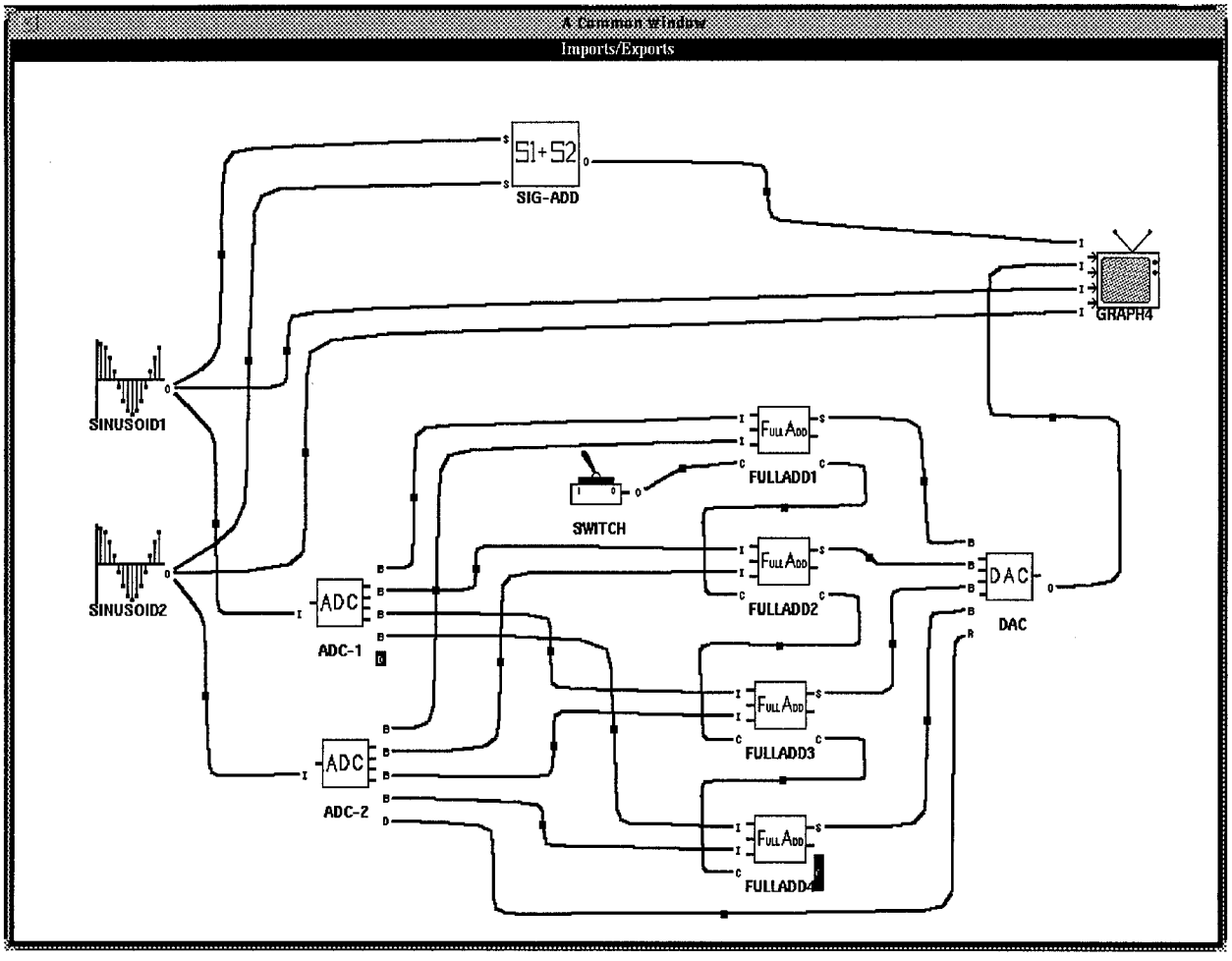


Figure 5.6 Multiple Domain Example

window for this application. The application contains two sinusoids (from the DSP domain) that are added together, resulting in an additional signal. The sinusoids are labeled "SINUSOID1" and "SINUSOID2" in Figure 5.6. When adding signals, the amplitudes of the two input signals are sampled simultaneously at various times. Each pair of samples is summed to generate the amplitude of the additional signal. This application adds the two sinusoids using two different methods as explained in the following sections.

5.5.1 First Method. The first method uses a signal-adder (from the DSP domain) to add the two sinusoids. The signal-adder is labeled "SIG-ADD" in Figure 5.6. The signal-adder, as modeled by Warner, simply adds each pair of samples from the two input signals, using ordinary arithmetic, to generate samples for the output signal. The signal-adder then supplies the resulting signal to the graph (from the DSP domain), labeled "GRAPH4." This method obviously generates an exact sum for each pair of samples.

5.5.2 Second Method. The second method uses two analog-to-digital converters (from the DSP domain) to sample the two sinusoids at the same times as the signal-adder. The converters are labeled "ADC-1" and "ADC-2" in Figure 5.6. At each sampling time, both analog-to-digital converters produce a four-bit code. The two-four bit codes are summed using four full-adders from the logic circuits domain, labeled "FULLADD1," "FULLADD2," "FULLADD3," and "FULLADD4." The least significant pair of bits generated by the analog-to-digital converters is summed by "FULLADD1." The second least significant pair of bits is summed by "FULLADD2," and so on. A switch from the logic circuits domain, labeled "SWITCH," is used supply the carry-in bit for "FULLADD1." The full-adders supply the sum for each pair of four-bit codes to a digital-to-analog converter

(from the DSP domain), labeled "DAC." The digital-to-analog converter then supplies the resulting signal to the graph. Unlike the first method, this method introduces a small amount of error when summing the sample pairs.

5.5.3 Key Observations. Figure 5.7 shows the results of the application. Four

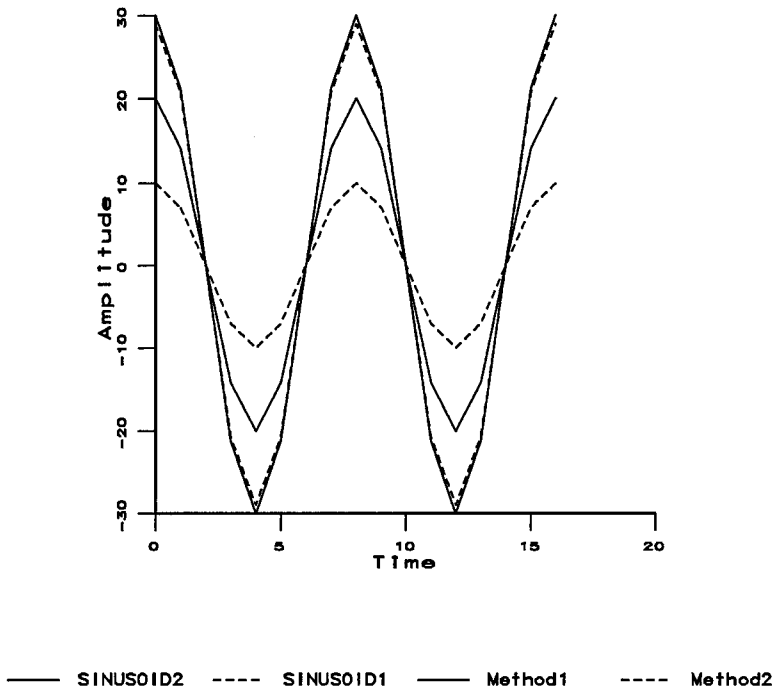


Figure 5.7 Sinusoid Plots

signals are displayed—the two sinusoids, plus the two signals generated by summing the sinusoids. The two sinusoids have identical frequencies and are in phase with each other. This makes it easy to visualize how the two additional signals should appear on the graph. The two sinusoids differ only by their maximum amplitudes. "SINUSOID1" has a maximum amplitude of 10.0 and "SINUSOID2" has a maximum amplitude of 20.0. Therefore, the two signals generated by summing the sinusoids should have maximum amplitudes of

30.0. As the graph indicates, they almost overlay each other. The solid plot represents the signal generated by the first method, while the dashed plot represents the signal generated by the second method. Again, the fact that they do not precisely overlay each other indicates a small amount of error was introduced by the analog-to-digital and digital-to-analog conversions.

As stated earlier, this application consolidates the testing and validation objectives. Primitives from three domains were actually included in this application because the full-adders belong to the "Circuits-Additional" domain and the switch belongs to the original logic circuits domain. In fact, there is no limit on the number domains that can be included in an application. The instance diagram in Figure 5.8 shows all subsystem and primitive

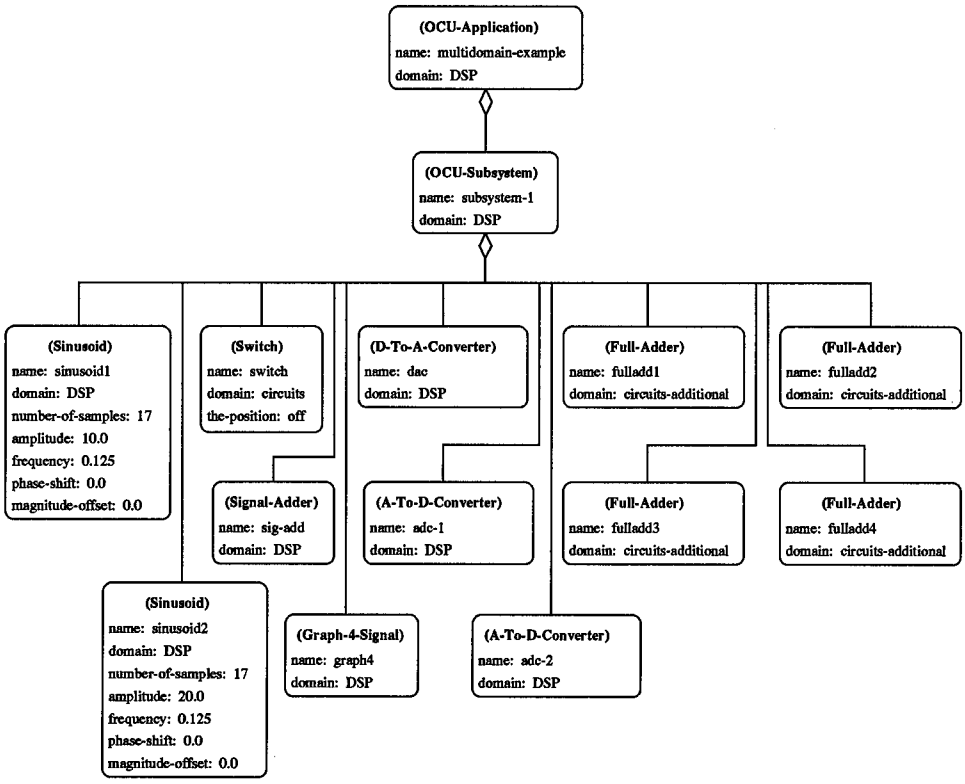


Figure 5.8 OCU-Application Instance Diagram

instances contained in the application. The name, domain, and other pertinent attributes for each instance are also identified.

This application was successfully executed, saved to the database, loaded from the database, and then successfully executed again. Also, this application required a primitive from an alternate domain to be edited. The switch, which supplies the carry-in to the first full-adder, had to be changed to the "off" position since its default position is "on." Finally, the accuracy of summing the sinusoids using two analog-to-digital converters, one digital-to-analog converter, and four full-adders demonstrates their correct implementation.

5.6 Summary

This chapter discussed the testing of Architect's multiple domain application capability. A set of primary and secondary test objectives was developed to ensure the new functionality of the Architect system and the new domain knowledge in the technology base were properly validated. Each of the objectives was satisfactorily demonstrated during testing. Finally, a substantial example of a multiple domain application was presented that demonstrated successful accomplishment of the test objectives.

VI. Conclusions and Recommendations

The objective of this research was to determine if OODBMS technology could be used to provide a multiple domain application capability for a domain-oriented application composition system. The original Architect system did not allow the composition of applications to cross domain boundaries—all primitives contained in a given application had to come from the same domain. This limitation is due to the constraints imposed by the software environment used to develop the Architect system. The file-based version of Architect uses this software environment to parse saved applications to and from the technology base. The parsing of these files is defined by grammars developed for the domains. Each domain has a separate and unique grammar. When parsing an application for one domain, knowledge of another domain's grammar is unavailable. Therefore, crossing domain boundaries was not possible with the file-based Architect system.

Previous research had already demonstrated that OODBMS technology could successfully provide the persistent technology base required by a system such as Architect. With the database version, Architect's applications are saved and loaded with a set of transformation functions, eliminating the need to parse files. Thus, the OODBMS technology opened the door for multiple domain application support.

Even after implementation of the OODBMS technology, there were obstacles associated with the design of the Architect system and the database transformation functions regarding multiple domain applications. Many of the activities associated with the composition, storage, retrieval, and execution of an application depend on information specific to a given domain. The design of the Architect system and database transformation functions

provided the capability to obtain this information from only one domain—the domain of the application being composed. Naturally, in the single domain application environment, this is sufficient because management of domain dynamics during the composition of an application is not required.

The concept of primary and alternate domains for application composition needed to be incorporated into Architect and the database transformation functions. A method was needed to manage the domain dynamics involved in the composition of multiple domain applications. A simple approach was taken. When performing certain activities associated with a primitive from an alternate domain, that alternate domain is temporarily established as the current domain. When the activity is completed, the primary domain is re-established as the current domain.

In addition to providing a management capability for domain dynamics, an additional domain needed to be implemented in the database. The DSP domain was selected for two reasons. First, it had already been implemented and validated in the file-based Architect system. Second, it was potentially compatible with the logic circuits domain, which was already implemented in the database. Both domains were enhanced with new primitives, such as the full-adder and analog-to-digital converter, to make them compatible. As a result, meaningful multiple domain applications could be composed, saved, loaded, edited, and executed.

6.1 Conclusions

Using an OODBMS such as ITASCA to store the persistent technology base of a domain-oriented application composition system such as Architect is the right choice.

The Architect system was designed using the object-oriented paradigm. Therefore, an OODBMS provides an excellent mapping of data between the persistent technology base and Architect's working technology base in REFINE. An OODBMS is much better at this than a flattened file structure. As this research has demonstrated, this advantage allowed more flexibility in managing the domain dynamics required in multiple domain applications.

A multiple domain application could not be achieved in the file-based version of Architect without a major redesign. The redesign would require that one large domain grammar be developed for all the domains. This approach would be poor design because domain knowledge would no longer be encapsulated for individual domains. This defies the logic of a "domain-oriented" system. Maintainability and extensibility would be negatively impacted because the modification or addition of just one domain would require the grammar for the entire system to be modified. The potential of introducing flaws into the system would increase. Because of these drawbacks, using OODBMS technology to achieve a multiple domain application capability is a better choice.

6.2 Recommendations for Improvement and Further Research

- **Provide Generic Support for Software Architectures:** Cecil and Fullenkamp provided generic support for domain definitions with implementation of their meta-model in the database. Similarly, generic support for software architectures should also be investigated and implemented in the database. This could result in the sharing of components across architecture boundaries, just as this thesis enabled the sharing of components across domain boundaries.

- **Validate the Database Version of Architect for Other Execution Modes:** Both the DSP and logic circuits domains implemented in the database operate in the non-event driven sequential mode. However, architect supports other execution modes. For example, Waggoner's (22) research led to the implementation of the cruise missile domain for execution in the time-driven sequential mode, and the logic circuits domain for execution in the event-driven sequential mode. Therefore, database support should be extended and validated for these execution modes.

- **Broaden the Scope of the Meta-Model for Domain Definitions:** The DSP domain contains several user-defined types. For example, "sample-type" is defined as a real number, and "real-signal-type" is defined as a sequence of real numbers. The meta-model does not contain a mechanism for incorporating this type of information. As a result, the REFINE source code automatically generated by the database was manually augmented to specify the user-defined types. An approach for automating this task through enhancement of the meta-model needs to be investigated.

- **Make Architect's Semantic Checks More Robust:** In order for two primitive's to be connected in an application, Architect's semantic checks require the connecting import and export areas to be of the same type and category. In this thesis, the binary signals of the digital-to-analog and analog-to-digital converters were given the type of "boolean" and the category of "signal" to be compatible with all components in the logic circuits domain. However, if the two domains had been developed independently, the converters very likely would have been assigned a category, such as "binary-signal," that does not precisely match the category of the logic circuits components. If that were the case, the components could not be connected, even though

they should be compatible. Techniques for making Architect's semantic checks more robust should be investigated to prevent this potential problem. Another approach would be to develop an interfacing mechanism allowing compatible components to communicate, even though they are not precisely categorized. Otherwise, domain engineers must redefine their domains to overcome this obstacle.

6.3 Final Comments

The Architect system has now demonstrated the capability of building multiple domain applications. OODBMS technology played a vital role in achieving this capability. The ability to cross domain boundaries during application composition gives the application specialist greater flexibility in developing applications. This provides for the possibility of reuse, a benefit often achieved in the object-oriented arena. There is no need to implement a given primitive in more than one domain—primitives can now be shared. The knowledge gained in this research can provide valuable insight for future generations of domain-oriented application composition systems as they are developed.

Appendix A. Sample Session: Multiple Domain Application

This appendix contains a script for an Architect session which builds a multiple domain application. The application contains primitives from both the logic circuits and DSP domains. This script is adapted from the sample session of Appendix E in Cecil's and Fullenkamp's thesis.

A.1 Start AVSI

REFINE must be loaded in an EMACS window. Once this is accomplished enter:

```
(load "dbl")
```

When the prompt returns, enter:

```
(dbl)
```

It will take several minutes for this file to run because it loads the DIALECT and INTERVISTA systems from the UNIX file system, and loads the Architect and AVSI files from the database. When the load is complete, the following prompt appears:

```
Load Complete
Type "(AVSI)" to start AVSI
```

Now enter the command:

```
(avsi)
```

This action loads the visual specification files for the domains currently defined for Architect. After the visual information is parsed into the object base, the control panel

appears in the upper left-hand corner of the screen. Across the top of the window is a row of icons that will be used to invoke many of the application composition functions of AVSI. The lower portion of the window is a message area used by AVSI to display status and error messages.

A.2 Create a New Application

1. Click any mouse button on the icon labeled CREATE NEW APPLICATION.
2. A pop-up window appears and prompts, SELECT DOMAIN. Click on the menu item DSP. At this time, the DSP domain definition is loaded from the database, as are the bitmap images for each primitive. DSP is the primary domain for this application.
3. A pop-up window appears with the prompt, ENTER NAME OF APPLICATION. Type

   ```
   multidomain-example
   ```

4. The name can be entered by hitting the "return" key or by clicking on DO IT at the bottom of the pop-up window.

A.3 Edit the Application

Now that the application has been created, the next step is to edit the application's make-up. Editing an application is composed of two separate operations: editing an application's components, and editing an application's update algorithm.

A.3.1 Add the Controlling Subsystem-Obj to the Application.

1. Click a mouse button on the EDIT APPLICATION control panel icon.

2. A pop-up menu appears with the prompt CHOOSE APPLICATION. Click on the menu item MULTIDOMAIN-EXAMPLE.

3. A pop-up menu appears with the prompt CHOOSE. Click on the menu item EDIT APPLICATION COMPONENTS. A blue window appears containing a single icon labeled,

   APPLICATION – OBJ
   MULTIDOMAIN – EX

4. Click on the diagram surface (anywhere on the blue surface except within the icon's boundary) of the window. A pop-up menu will appear.

5. Select CREATE NEW SUBSYSTEM.

6. A pop-up window appears, with the prompt ENTER A NAME. Enter `multi-sub`

7. A box outline of an icon appears, attached to the mouse cursor. Place the icon below the application-obj icon by moving the cursor to the desired location and clicking.

8. Click any mouse button on the newly created subsystem-obj icon and select the menu option LINK TO SOURCE.

9. The mouse cursor changes from an arrow to an oval with a dot in it, signifying that an object needs to be selected. Place the mouse cursor on the application-obj's icon and click any mouse button. A link appears between the application-obj's icon and the subsystem-obj's icon.

10. Close the edit-application-window by clicking on the diagram surface and selecting DEACTIVATE.

A.3.2 Create the Application-Obj's Update Algorithm.

1. Click a mouse button on the EDIT APPLICATION control panel icon.

2. A pop-up menu appears with the prompt CHOOSE APPLICATION. Click on the menu item MULTIDOMAIN-EXAMPLE.

3. A pop-up menu appears with the prompt CHOOSE. Click on the menu item EDIT APPLICATION UPDATE. Three windows will appear. One contains a graphical view of the update algorithm, one contains a textual view of the algorithm, and the third (the Controllee Window) shows the icons that represent the application-obj's controllees (with two extra icons for if-then-else and while-do constructs). The graphical update window contains two icons, "Start" and "End", with a dotted arrow pointing from the start-icon to the end-icon.

4. Click a mouse button on the icon in the controllee window labeled $\begin{matrix}\text{SUBSYSTEM - OBJ}\\ \text{MULTI - SUB}\end{matrix}$. The cursor changes to an oval with a dot in it indicating that an object needs to be selected.

5. Click on the "nub" on the dotted line midway between the start and end icons. This will cause the update sequence to redraw with the subsystem-obj included. Note the textual representation is automatically updated to reflect each change in the diagram window.

6. Close the edit-update-algorithm windows by clicking on the black title bar at the top of the graphical update window and selecting DEACTIVATE.

A.4 Edit the Subsystems

Building a subsystem is similar to building the application. This section illustrates how to instantiate primitive objects and link them to the controlling subsystem created in the previous section.

The subsystem, MULTI-SUB, will control a sinusoid, an analog-to-digital converter and four LEDs. To add these objects, perform the following steps:

A.4.1 Add the Primitive Objects.

1. Click on the EDIT SUBSYSTEM icon in the control panel window.
2. Click on the menu item MULTI-SUB. A white window opens (the subsystem window) which contains an OCU representation of a subsystem.
3. Click on the OBJECTS icon in the subsystem window. The blue edit-subsystem-window for MULTI-SUB appears, containing a single icon labeled $\frac{\text{SUBSYSTEM - OBJ}}{\text{MULTI - SUB}}$. A green window, the technology-base window, also appears and contains an icon for each primitive-object in the current domain, DSP.
4. Click on the icon in the green technology-base-window labeled SINSOID.
5. A Sinusoid-icon is created and attached to the mouse cursor. Place this icon on the blue edit-subsystem-window near MULTI-SUB.
6. Name the sinusoid by typing THE-SINUSOID in the pop-up window.
7. Similarily, create an analog-to-digital converter object named THE-ADC.
8. Next, obtain the technology base window for the logic circuits domain. Click on the diagram surface of the technology base window. A pop-up menu will appear.

9. Select GET-PRIMITIVES-FROM-AN-ALTERNATE-DOMAIN. Another pop-up menu will appear.

10. Select CIRCUITS. The technology base window for the logic circuits domain will appear. Create four instances of the LED object named LED1, LED2, LED3, and LED4.

11. Link the primitive objects to MULTI-SUB by clicking a mouse button on MULTI-SUB, and selecting LINK MULTIPLE TARGETS from the pop-up menu.

12. A pop-up window will appear that lists all the unconnected objects in the edit-subsystem-window. Select ALL OF THE ABOVE, and click on DO IT. A link will appear from MULTI-SUB to each of the other icons.

13. Close the edit-subsystem-window and the technology-base-window by clicking on the blue surface and selecting DEACTIVATE. The windows can be closed separately by selecting DEACTIVATE from their title bar menus.

    At this point, the subsystem window for MULTI-SUB will again be visible.

A.4.2 Connect Imports and Exports. To connect the import and export objects perform the following steps:

1. Click a mouse button on the IMPORT AREA or EXPORT AREA icon in MULTI-SUB's subsystem window.

2. Select MAKE CONNECTIONS from the pop-up window. A red window (the imports-exports window) will open and contain the sinusoid icon, the analog-to-digital converter icon, and the four LED icons. The black bars (these bars are actually high-

lighted subicons attached to the primitive's icon) on the sides of the primitive icons indicate connections that need to be made.

3. Icons can be moved to new positions on the screen by clicking on the icon and selecting MOVE ICON from the pop-up menu. A square grid, attached to the mouse, appears. Move the grid to the new icon position and click to "drop" the icon.

4. Connect sinusoid THE-SINUSOID to analog-to-digital converter THE-ADC by clicking on the black bar of THE-SINUSOID and then clicking on the black bar on the left side of THE-ADC. Connect BIT1 of THE-ADC to LED LED1. Connect BIT2 of THE-ADC to LED LED2. Connect BIT3 of THE-ADC to LED LED3. Connect BIT3 of THE-ADC to LED LED4.

5. Close the imports-exports window by clicking on the red surface and selecting DEACTIVATE, or by selecting DEACTIVATE from each window's title bar menu.

A.4.3 Build MULTI-SUB*'s Update Algorithm.* After the imports-exports window has been closed, the white subsystem window will again be visible. Building the update algorithm for MULTI-SUB is similar to building the update algorithm for the application, and requires the following steps:

1. Click the mouse on the CONTROLLER icon in the subsystem window. The three windows that were seen before are exposed, except the controllee window now contains the sinusoid, the analog-to-digital converter, and the four LEDs controlled by MULTI-SUB.

2. Add each controllee to the update sequence by clicking on the controllee icon and then clicking on the "nub" in the graphical update window that represents the proper sequence position for the controllee. The order in which the controllees must appear is:

```
THE-SINUSOID    THE-ADC    LED1    LED2    LED3    LED4
```

   Note that the textual update description is updated as the graphical update is built.

3. Close the windows by clicking on the graphical update window title bar and selecting DEACTIVATE.

A.5 Perform Semantic Checks

Semantic checks are performed by Architect as part of the imports-exports connection process. However, the semantic checks may be run at any time by clicking on the control panel icon labeled CHECK SEMANTICS. The results of the semantic checks may be viewed in the EMACS window.

A.6 Execute the Application

Now that the application has been fully defined, it can be executed. Each time the application is executed, the analog-to-digital converter processes one sample of the sinusoid. The four LED objects identify the value of the four-bit code that is generated by the converter. Click on the control panel button labeled EXECUTE APPLICATION. The results are displayed in the EMACS window.

Appendix B. REFINE *Update Functions*

This appendix contains the REFINE source code for the update functions of all primitives developed for this thesis.

B.1 And-Gate-3Input

```
function AND-GATE-3INPUT-UPDATE (subsystem : subsystem-obj,
                           and-gate-3input : AND-GATE-3INPUT) =

  format(debug-on, \"AND-GATE-3INPUT-UPDATE on ~s~%\", name(and-gate-3input));

  let (in1 : boolean = get-import('in1, subsystem, and-gate-3input),
       in2 : boolean = get-import('in2, subsystem, and-gate-3input),
       in3 : boolean = get-import('in3, subsystem, and-gate-3input))

  set-export(subsystem, and-gate-3input, 'out1, (in1 & in2 & in3))
```

B.2 Or-Gate-3Input

```
function OR-GATE-3INPUT-UPDATE (subsystem : subsystem-obj,
                           or-gate-3input : OR-GATE-3INPUT) =

  format(debug-on, \"OR-GATE-3INPUT-UPDATE on ~s~%\", name(or-gate-3input));

  let (in1 : boolean = get-import('in1, subsystem, or-gate-3input),
       in2 : boolean = get-import('in2, subsystem, or-gate-3input),
       in3 : boolean = get-import('in3, subsystem, or-gate-3input))

  set-export(subsystem, or-gate-3input, 'out1, (in1 or in2 or in3))
```

B.3 Full-Adder

```
function FULL-ADDER-UPDATE (subsystem  : subsystem-obj,
                            full-adder : FULL-ADDER) =

  format(debug-on, \"FULL-ADDER-UPDATE on ~s~%\", name(full-adder));

 let (in1      : boolean = get-import('in1,      subsystem, full-adder),
      in2      : boolean = get-import('in2,      subsystem, full-adder),
      carry-in : boolean = get-import('carry-in, subsystem, full-adder))
```

```
if ~in1 and ~in2 and ~carry-in then
  set-export(subsystem, full-adder, 'sum, nil);
  set-export(subsystem, full-adder, 'carry-out, nil)
elseif in1 and ~in2 and ~carry-in then
  set-export(subsystem, full-adder, 'sum, true);
  set-export(subsystem, full-adder, 'carry-out, nil)
elseif ~in1 and in2 and ~carry-in then
  set-export(subsystem, full-adder, 'sum, true);
  set-export(subsystem, full-adder, 'carry-out, nil)
elseif in1 and in2 and ~carry-in then
  set-export(subsystem, full-adder, 'sum, nil);
  set-export(subsystem, full-adder, 'carry-out, true)
elseif ~in1 and ~in2 and carry-in then
  set-export(subsystem, full-adder, 'sum, true);
  set-export(subsystem, full-adder, 'carry-out, nil)
elseif in1 and ~in2 and carry-in then
  set-export(subsystem, full-adder, 'sum, nil);
  set-export(subsystem, full-adder, 'carry-out, true)
elseif ~in1 and in2 and carry-in then
  set-export(subsystem, full-adder, 'sum, nil);
  set-export(subsystem, full-adder, 'carry-out, true)
elseif in1 and in2 and carry-in then
  set-export(subsystem, full-adder, 'sum, true);
  set-export(subsystem, full-adder, 'carry-out, true)
```

B.4 Full-Adder-4Bit

```
function FULL-ADDER-4BIT-UPDATE (subsystem  : subsystem-obj,
                                 full-adder-4bit : FULL-ADDER-4BIT) =

 format(debug-on, \"FULL-ADDER-4BIT-UPDATE on ~s~%\", name(full-adder-4bit));

let (in-1a : boolean = get-import('in-1a, subsystem, full-adder-4bit),
     in-1b : boolean = get-import('in-1b, subsystem, full-adder-4bit),
     in-2a : boolean = get-import('in-2a, subsystem, full-adder-4bit),
     in-2b : boolean = get-import('in-2b, subsystem, full-adder-4bit),
     in-3a : boolean = get-import('in-3a, subsystem, full-adder-4bit),
     in-3b : boolean = get-import('in-3b, subsystem, full-adder-4bit),
     in-4a : boolean = get-import('in-4a, subsystem, full-adder-4bit),
     in-4b : boolean = get-import('in-4b, subsystem, full-adder-4bit),
     temp-carry : boolean = true)

(if ~in-1a and ~in-1b then
  set-export(subsystem, full-adder-4bit, 'out-1c, nil);
  temp-carry <- false
elseif in-1a and in-1b then
  set-export(subsystem, full-adder-4bit, 'out-1c, nil);
  temp-carry <- true
```

```
else
  set-export(subsystem, full-adder-4bit, 'out-1c, true);
  temp-carry <- false);

(if ~in-2a and ~in-2b and ~temp-carry then
  set-export(subsystem, full-adder-4bit, 'out-2c, nil);
  temp-carry <- false
elseif in-2a and in-2b and temp-carry then
  set-export(subsystem, full-adder-4bit, 'out-2c, true);
  temp-carry <- true
elseif (~in-2a and in-2b and temp-carry) or
       (in-2a and ~in-2b and temp-carry) or
       (in-2a and in-2b and ~temp-carry) then
  set-export(subsystem, full-adder-4bit, 'out-2c, nil);
  temp-carry <- true
else
  set-export(subsystem, full-adder-4bit, 'out-2c, true);
  temp-carry <- false);

(if ~in-3a and ~in-3b and ~temp-carry then
  set-export(subsystem, full-adder-4bit, 'out-3c, nil);
  temp-carry <- false
elseif in-3a and in-3b and temp-carry then
  set-export(subsystem, full-adder-4bit, 'out-3c, true);
  temp-carry <- true
elseif (~in-3a and in-3b and temp-carry) or
       (in-3a and ~in-3b and temp-carry) or
       (in-3a and in-3b and ~temp-carry) then
  set-export(subsystem, full-adder-4bit, 'out-3c, nil);
  temp-carry <- true
else
  set-export(subsystem, full-adder-4bit, 'out-3c, true);
  temp-carry <- false);

(if ~in-4a and ~in-4b and ~temp-carry then
  set-export(subsystem, full-adder-4bit, 'out-4c, nil);
  temp-carry <- false
elseif in-4a and in-4b and temp-carry then
  set-export(subsystem, full-adder-4bit, 'out-4c, true);
  temp-carry <- true
elseif (~in-4a and in-4b and temp-carry) or
       (in-4a and ~in-4b and temp-carry) or
       (in-4a and in-4b and ~temp-carry) then
  set-export(subsystem, full-adder-4bit, 'out-4c, nil);
  temp-carry <- true
else
  set-export(subsystem, full-adder-4bit, 'out-4c, true);
  temp-carry <- false);

(if temp-carry then
  set-export(subsystem, full-adder-4bit, 'carry-out, true)
```

```
else
  set-export(subsystem, full-adder-4bit, 'carry-out, nil))
```

B.5 Full-Subtractor

```
function FULL-SUBTRACTOR-UPDATE (subsystem  : subsystem-obj,
                                 full-subtractor : FULL-SUBTRACTOR) =

 format(debug-on, \"FULL-SUBTRACTOR-UPDATE on ~s~%\", name(full-subtractor));

 let (minuend         : boolean = get-import('minuend,         subsystem, full-subtractor),
      subtrahend      : boolean = get-import('subtrahend,      subsystem, full-subtractor),
      previous-borrow : boolean = get-import('previous-borrow, subsystem, full-subtractor))

 if ~minuend and ~subtrahend and ~previous-borrow then
   set-export(subsystem, full-subtractor, 'difference, nil);
   set-export(subsystem, full-subtractor, 'borrow-bit, nil)
 elseif minuend and ~subtrahend and ~previous-borrow then
   set-export(subsystem, full-subtractor, 'difference, true);
   set-export(subsystem, full-subtractor, 'borrow-bit, nil)
 elseif ~minuend and subtrahend and ~previous-borrow then
   set-export(subsystem, full-subtractor, 'difference, true);
   set-export(subsystem, full-subtractor, 'borrow-bit, true)
 elseif minuend and subtrahend and ~previous-borrow then
   set-export(subsystem, full-subtractor, 'difference, nil);
   set-export(subsystem, full-subtractor, 'borrow-bit, nil)
 elseif ~minuend and ~subtrahend and previous-borrow then
   set-export(subsystem, full-subtractor, 'difference, true);
   set-export(subsystem, full-subtractor, 'borrow-bit, true)
 elseif minuend and ~subtrahend and previous-borrow then
   set-export(subsystem, full-subtractor, 'difference, nil);
   set-export(subsystem, full-subtractor, 'borrow-bit, nil)
 elseif ~minuend and subtrahend and previous-borrow then
   set-export(subsystem, full-subtractor, 'difference, nil);
   set-export(subsystem, full-subtractor, 'borrow-bit, true)
 elseif minuend and subtrahend and previous-borrow then
   set-export(subsystem, full-subtractor, 'difference, true);
   set-export(subsystem, full-subtractor, 'borrow-bit, true)
```

B.6 Full-Subtractor-4Bit

```
function FULL-SUBTRACTOR-4BIT-UPDATE (subsystem  : subsystem-obj,
                                      full-subtractor-4bit : FULL-SUBTRACTOR-4BIT) =

  format(debug-on, \"FULL-SUBTRACTOR-4BIT-UPDATE on ~s~%\", name(full-subtractor-4bit));
```

```
let (minuend-1    : boolean = get-import('minuend-1,    subsystem, full-subtractor-4bit),
     minuend-2    : boolean = get-import('minuend-2,    subsystem, full-subtractor-4bit),
     minuend-3    : boolean = get-import('minuend-3,    subsystem, full-subtractor-4bit),
     minuend-4    : boolean = get-import('minuend-4,    subsystem, full-subtractor-4bit),
     subtrahend-1 : boolean = get-import('subtrahend-1, subsystem, full-subtractor-4bit),
     subtrahend-2 : boolean = get-import('subtrahend-2, subsystem, full-subtractor-4bit),
     subtrahend-3 : boolean = get-import('subtrahend-3, subsystem, full-subtractor-4bit),
     subtrahend-4 : boolean = get-import('subtrahend-4, subsystem, full-subtractor-4bit),
     temp-borrow  : boolean = true)

(if minuend-1 and ~subtrahend-1 then
  set-export(subsystem, full-subtractor-4bit, 'difference-1, true);
  temp-borrow <- false
elseif ~minuend-1 and subtrahend-1 then
  set-export(subsystem, full-subtractor-4bit, 'difference-1, true);
  temp-borrow <- true
else
  set-export(subsystem, full-subtractor-4bit, 'difference-1, nil);
  temp-borrow <- false);

(if ~minuend-2 and subtrahend-2 and temp-borrow then
  set-export(subsystem, full-subtractor-4bit, 'difference-2, nil);
  temp-borrow <- true
elseif minuend-2 and ~subtrahend-2 and ~temp-borrow then
  set-export(subsystem, full-subtractor-4bit, 'difference-2, true);
  temp-borrow <- false
elseif (~minuend-2 and ~subtrahend-2 and ~temp-borrow) or
       (minuend-2 and ~subtrahend-2 and temp-borrow) or
       (minuend-2 and subtrahend-2 and ~temp-borrow) then
  set-export(subsystem, full-subtractor-4bit, 'difference-2, nil);
  temp-borrow <- false
else
  set-export(subsystem, full-subtractor-4bit, 'difference-2, true);
  temp-borrow <- true);

(if ~minuend-3 and subtrahend-3 and temp-borrow then
  set-export(subsystem, full-subtractor-4bit, 'difference-3, nil);
  temp-borrow <- true
elseif minuend-3 and ~subtrahend-3 and ~temp-borrow then
  set-export(subsystem, full-subtractor-4bit, 'difference-3, true);
  temp-borrow <- false
elseif (~minuend-3 and ~subtrahend-3 and ~temp-borrow) or
       (minuend-3 and ~subtrahend-3 and temp-borrow) or
       (minuend-3 and subtrahend-3 and ~temp-borrow) then
  set-export(subsystem, full-subtractor-4bit, 'difference-3, nil);
  temp-borrow <- false
else
  set-export(subsystem, full-subtractor-4bit, 'difference-3, true);
  temp-borrow <- true);

(if ~minuend-4 and subtrahend-4 and temp-borrow then
```

```
  set-export(subsystem, full-subtractor-4bit, 'difference-4, nil);
  temp-borrow <- true
elseif minuend-4 and ~subtrahend-4 and ~temp-borrow then
  set-export(subsystem, full-subtractor-4bit, 'difference-4, true);
  temp-borrow <- false
elseif (~minuend-4 and ~subtrahend-4 and ~temp-borrow) or
       (minuend-4 and ~subtrahend-4 and temp-borrow) or
       (minuend-4 and subtrahend-4 and ~temp-borrow) then
  set-export(subsystem, full-subtractor-4bit, 'difference-4, nil);
  temp-borrow <- false
else
  set-export(subsystem, full-subtractor-4bit, 'difference-4, true);
  temp-borrow <- true);

(if temp-borrow then
  set-export(subsystem, full-subtractor-4bit, 'borrow-bit, true)
else
  set-export(subsystem, full-subtractor-4bit, 'borrow-bit, nil))
```

B.7 Analog-To-Digital Converter

```
function A-TO-D-CONVERTER-UPDATE (subsystem : subsystem-obj,
                        the-a-to-d-converter : A-TO-D-CONVERTER) =

  format(dsp-debug, \"A-TO-D-CONVERTER-UPDATE on ~s~%\", name(the-a-to-d-converter));

  let (S : real-signal-type = a-to-d-converter-holder(the-a-to-d-converter),
       MM : real = a-to-d-converter-max-magnitude(the-a-to-d-converter),
       LowValue : real = 0.0 - a-to-d-converter-max-magnitude(the-a-to-d-converter),
       temp-bits : seq(boolean) = [true, nil, nil, true],
       LevelDelta : real = 2 * a-to-d-converter-max-magnitude(the-a-to-d-converter) / 15.0)

  (if a-to-d-converter-restart(the-a-to-d-converter) then
    S <- get-import('input, subsystem, the-a-to-d-converter);
    a-to-d-converter-holder(the-a-to-d-converter) <- S;
    a-to-d-converter-restart(the-a-to-d-converter) <- false);

  S(1) > (LowValue + LevelDelta * 1.0) --> (temp-bits <- [nil, true, nil, true]);

  S(1) > (LowValue + LevelDelta * 2.0) --> (temp-bits <- [true, true, nil, true]);

  S(1) > (LowValue + LevelDelta * 3.0) --> (temp-bits <- [nil, nil, true, true]);

  S(1) > (LowValue + LevelDelta * 4.0) --> (temp-bits <- [true, nil, true, true]);

  S(1) > (LowValue + LevelDelta * 5.0) --> (temp-bits <- [nil, true, true, true]);

  S(1) > (LowValue + LevelDelta * 6.0) --> (temp-bits <- [true, true, true, true]);
```

```
  S(1) > (LowValue + LevelDelta * 7.0) --> (temp-bits <- [nil, nil, nil, nil]);

  S(1) > (LowValue + LevelDelta * 8.0) --> (temp-bits <- [true, nil, nil, nil]);

  S(1) > (LowValue + LevelDelta * 9.0) --> (temp-bits <- [nil, true, nil, nil]);

  S(1) > (LowValue + LevelDelta * 10.0) --> (temp-bits <- [true, true, nil, nil]);

  S(1) > (LowValue + LevelDelta * 11.0) --> (temp-bits <- [nil, nil, true, nil]);

  S(1) > (LowValue + LevelDelta * 12.0) --> (temp-bits <- [true, nil, true, nil]);

  S(1) > (LowValue + LevelDelta * 13.0) --> (temp-bits <- [nil, true, true, nil]);

  S(1) > (LowValue + LevelDelta * 14.0) --> (temp-bits <- [true, true, true, nil]);

  S(1) > MM or S(1) < LowValue -->
     format(dsp-debug, \"Sample is out of range; change upper or lower limit\");

  set-export(subsystem, the-a-to-d-converter, 'bit1, temp-bits(1));
  set-export(subsystem, the-a-to-d-converter, 'bit2, temp-bits(2));
  set-export(subsystem, the-a-to-d-converter, 'bit3, temp-bits(3));
  set-export(subsystem, the-a-to-d-converter, 'bit4, temp-bits(4));

  S <- rest(S);
  a-to-d-converter-holder(the-a-to-d-converter) <- S;
  set-export(subsystem, the-a-to-d-converter, 'done, false);
  (if size(S) = 0 then
    a-to-d-converter-restart(the-a-to-d-converter) <- true;
    set-export(subsystem, the-a-to-d-converter, 'done, true))
```

B.8 Digital-To-Analog Converter

```
function D-TO-A-CONVERTER-UPDATE (subsystem : subsystem-obj,
                        the-d-to-a-converter : D-TO-A-CONVERTER) =

  format(dsp-debug, \"D-TO-A-CONVERTER-UPDATE on ~s~%\", name(the-d-to-a-converter));

  let (S : real-signal-type = d-to-a-converter-holder(the-d-to-a-converter),
       MM : real = d-to-a-converter-max-magnitude(the-d-to-a-converter),
       LevelDelta : real = 2.0 * d-to-a-converter-max-magnitude(the-d-to-a-converter) / 15.0,
       bit1 : boolean = get-import('bit1, subsystem, the-d-to-a-converter),
       bit2 : boolean = get-import('bit2, subsystem, the-d-to-a-converter),
       bit3 : boolean = get-import('bit3, subsystem, the-d-to-a-converter),
       bit4 : boolean = get-import('bit4, subsystem, the-d-to-a-converter),
       release : boolean = get-import('release, subsystem, the-d-to-a-converter))

  (if size(S) = 0 then
    set-export(subsystem, the-d-to-a-converter, 'output, S));
```

```
(if bit4 and ~bit3 and ~bit2 and bit1 then
   S <- append(S, (0.0 - MM + 0.5 * LevelDelta))
 elseif bit4 and ~bit3 and bit2 and ~bit1 then
   S <- append(S, (0.0 - MM + 1.5 * LevelDelta))
 elseif bit4 and ~bit3 and bit2 and bit1 then
   S <- append(S, (0.0 - MM + 2.5 * LevelDelta))
 elseif bit4 and bit3 and ~bit2 and ~bit1 then
   S <- append(S, (0.0 - MM + 3.5 * LevelDelta))
 elseif bit4 and bit3 and ~bit2 and bit1 then
   S <- append(S, (0.0 - MM + 4.5 * LevelDelta))
 elseif bit4 and bit3 and bit2 and ~bit1 then
   S <- append(S, (0.0 - MM + 5.5 * LevelDelta))
 elseif bit4 and bit3 and bit2 and bit1 then
   S <- append(S, (0.0 - MM + 6.5 * LevelDelta))
 elseif ~bit4 and ~bit3 and ~bit2 and ~bit1 then
   S <- append(S, (0.0 - MM + 7.5 * LevelDelta))
 elseif ~bit4 and ~bit3 and ~bit2 and bit1 then
   S <- append(S, (0.0 - MM + 8.5 * LevelDelta))
 elseif ~bit4 and ~bit3 and bit2 and ~bit1 then
   S <- append(S, (0.0 - MM + 9.5 * LevelDelta))
 elseif ~bit4 and ~bit3 and bit2 and bit1 then
   S <- append(S, (0.0 - MM + 10.5 * LevelDelta))
 elseif ~bit4 and bit3 and ~bit2 and ~bit1 then
   S <- append(S, (0.0 - MM + 11.5 * LevelDelta))
 elseif ~bit4 and bit3 and ~bit2 and bit1 then
   S <- append(S, (0.0 - MM + 12.5 * LevelDelta))
 elseif ~bit4 and bit3 and bit2 and ~bit1 then
   S <- append(S, (0.0 - MM + 13.5 * LevelDelta))
 elseif ~bit4 and bit3 and bit2 and bit1 then
   S <- append(S, (0.0 - MM + 14.5 * LevelDelta)));

d-to-a-converter-holder(the-d-to-a-converter) <- S;

(if release then
  set-export(subsystem, the-d-to-a-converter, 'output, S);
  d-to-a-converter-holder(the-d-to-a-converter) <- [])
```

Appendix C. Object Model Diagrams for the Digital Signal Processing Domain

This appendix contains the diagrams for the object model of the digital signal processing domain.

C.1 Abstract Classes of DSP

Figure C.1 shows the abstract classes in the DSP domain.

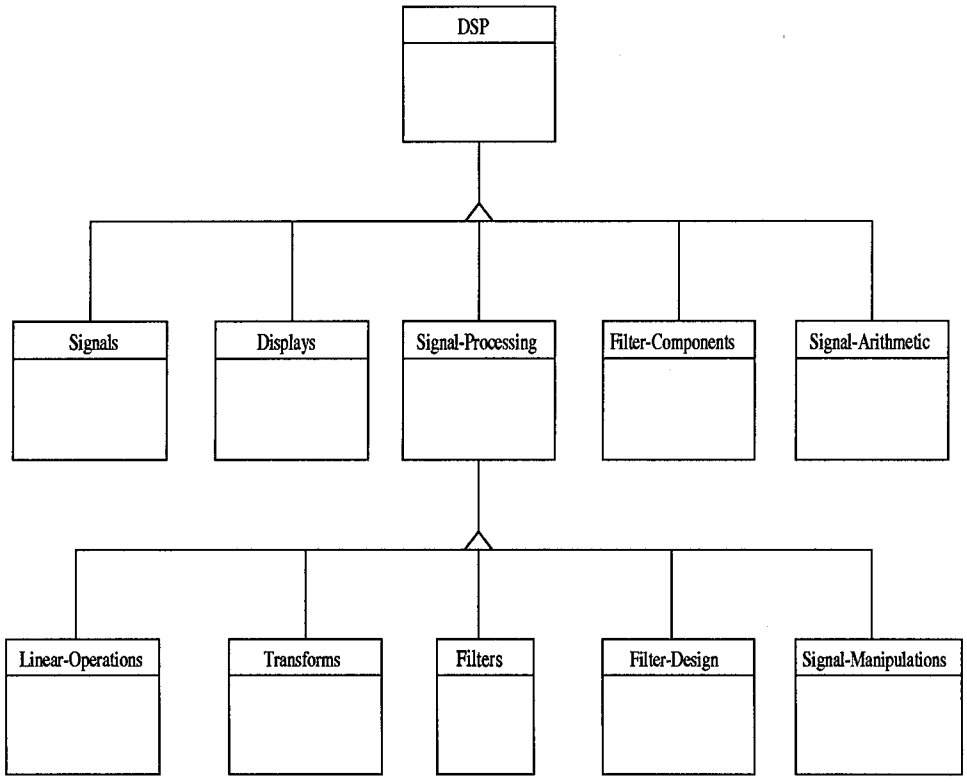


Figure C.1 Abstract Classes of DSP Domain

C.2 Concrete Subclasses of "Signals"

Figure C.2 shows the concrete subclasses of the "Signals" abstract class.

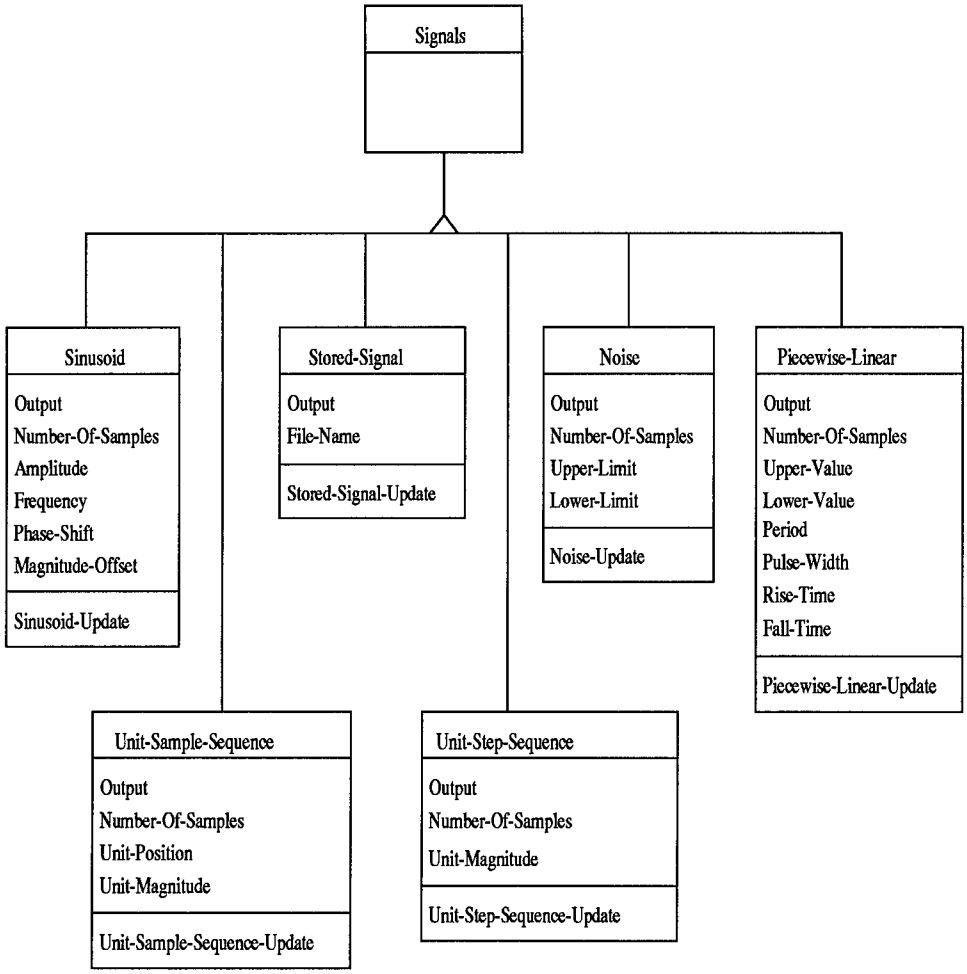


Figure C.2 Concrete Subclasses of "Signals"

C.3 Concrete Subclasses of "Displays"

Figure C.3 shows the concrete subclasses of the "Displays" abstract class.

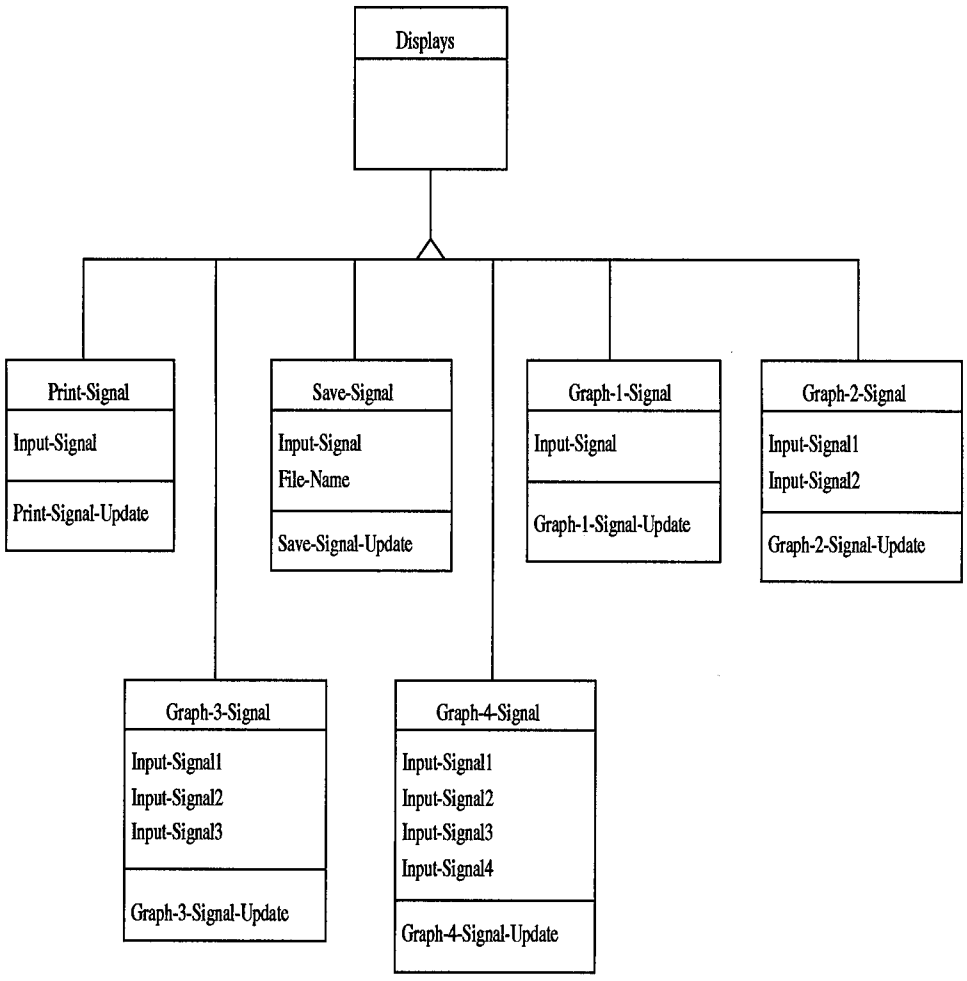


Figure C.3 Concrete Subclasses of "Displays"

C.4 Concrete Subclasses of "Filter Components"

Figure C.4 shows the concrete subclasses of the "Filter Components" abstract class.

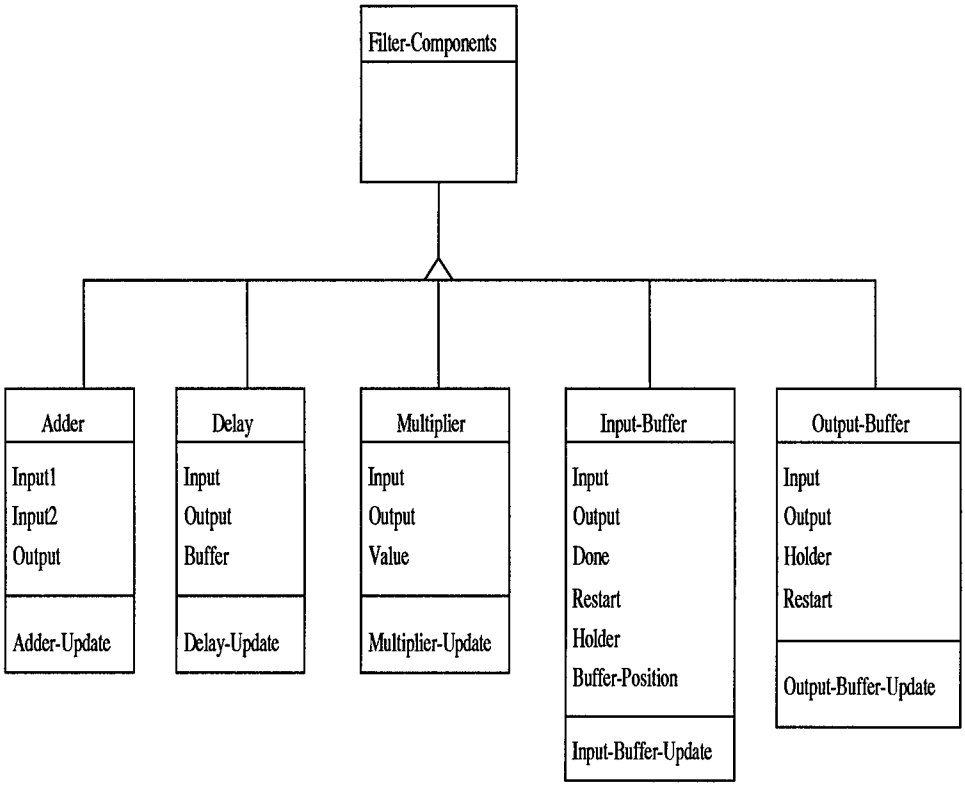


Figure C.4 Concrete Subclasses of "Filter Components"

C.5 Concrete Subclasses of "Signal Arithmetic"

Figure C.5 shows the concrete subclasses of the "Signal Arithmetic" abstract class.

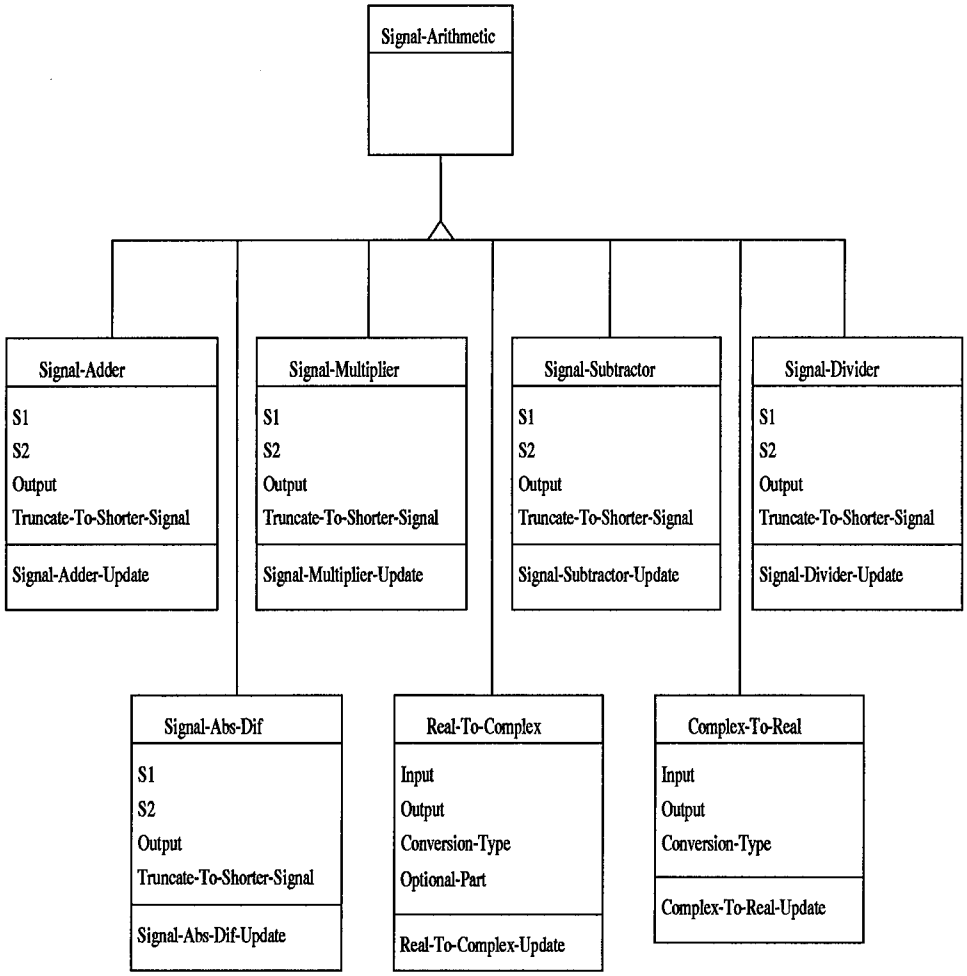


Figure C.5 Concrete Subclasses of "Signal Arithmetic"

C.6 Concrete Subclasses of "Signal Processing"

Figure C.6 shows the concrete subclasses of the "Signal Processing" abstract class.

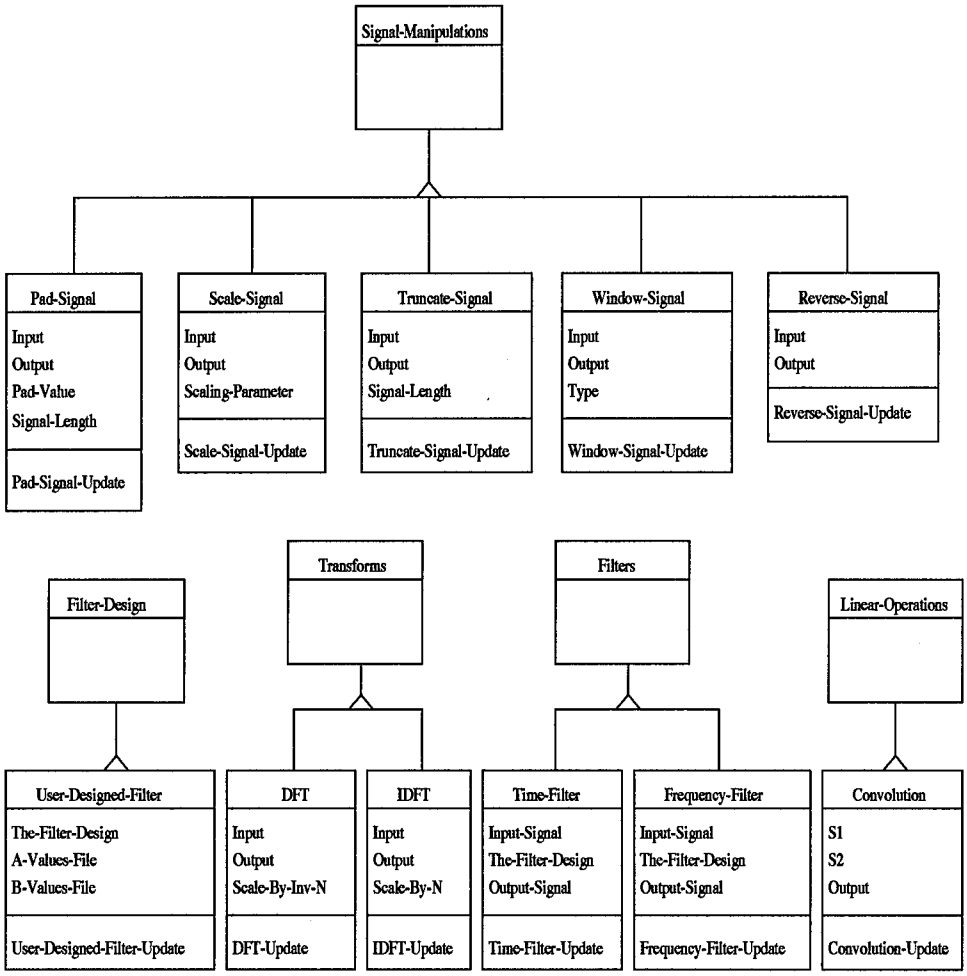


Figure C.6 Concrete Subclasses of "Signal Processing"

Bibliography

1. Ahmed, Shamim and others. "Object-Oriented Database Management Systems for Engineering: A Comparison," *Journal of Object-Oriented Programming*, *5*:27–44 (June 1992).
2. Anderson, Cynthia. *Creating and Manipulating Formalized Software Architectures in Support of a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D-01, Air Force Institute of Technology, December 1992.
3. Atwood, Thomas M. "The Case for Object-Oriented Databases," *IEEE Spectrum*, *28*:44–47 (February 1991).
4. Bailor, Paul D. and others. "An Integrated Technology Approach to the Development of Software Composition Systems." *Computers in Engineering Conference.* (in press). 1995.
5. Bertino, Elisa and Lorenzo Martino. "Object-Oriented Database Management Systems: Concepts and Issues," *IEEE Computer*, *24*:33–47 (April 1991).
6. Cattell, R. G. G. *Object Data Management.* Englewood Cliffs, New Jersey: Prentice Hall, 1991.
7. Cecil, Danny A. and Joseph A. Fullenkamp. *Using Database Technology to Support Domain-Oriented Application Composition Systems.* MS thesis, AFIT/GCS/ENG/93D-03, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1993.
8. Cossentine, Jay A. *Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D-04, Air Force Institute of Technology, December 1993.
9. Franz, Inc. *Allegro COMMON WINDOWS on X Manual.* Berkeley, CA, August 1990.
10. Garlan, David and Mary Shaw. "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, *I* (1993).
11. Gool, Warren Evan. *Alternative Architectures for Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D-11, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1993.
12. Joseph, John V. and others. "Object-Oriented Databases: Design and Implementation," *Proceedings of the IEEE*, *79*:42–64 (January 1991).
13. Korth, Henry F. and Abraham Silberschatz. *Database System Concepts.* New York: McGraw-Hill, 1991.
14. Lee, Kenneth J. and others. *Model-Based Software Development (Draft).* Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.
15. Mano, M. Morris. *Digital Logic and Computer Design.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1979.

16. Oppenheim, Alan V. and Ronald W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall, 1989.

17. Randour, Mary Anne. *Creating and Manipulating a Domain-Specific Formal Object Base*. MS thesis, AFIT/GCS/ENG/92D-13, Air Force Institute of Technology, December 1992.

18. Reasoning Systems, Inc. *DIALECT User's Guide*. Palo Alto, CA, July 1990.

19. Reasoning Systems, Inc. *REFINE User's Guide*. Palo Alto, CA, May 1990.

20. Reasoning Systems, Inc. *INTERVISTA User's Guide*. Palo Alto, CA, March 1991.

21. Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

22. Waggoner, Robert W. *Domain Modeling of Time-Dependent Systems*. MS thesis, AFIT/GCS/ENG/93D-23, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1993.

23. Warner, Russell Mark. *A Method for Populating the Knowledge Base of AFIT's Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93D-24, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1993.

24. Weide, Timothy. *Development of a Visual System for a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93M-05, Air Force Institute of Technology, March 1993.

Vita

Captain Alfred W. Harris Jr was born April 8, 1959 in Shelby, North Carolina and graduated from East Rutherford High School in Forest City, North Carolina in 1977. He earned a Bachelor of Science degree in Civil Engineering from North Carolina State University in 1982. In April 1984, he was commissioned through Officer Training School, Lackland AFB, Texas. After receiving a year of technical training at Keesler AFB, Mississippi, he served as a programmer/analyst on several command and control systems from April 1985 to October 1989 at the Command and Control Systems Center, Tinker AFB, Oklahoma. He managed the sizing and performance analyses of Air Force standard base level computers from October 1989 to May 1993 while assigned to the Standard Systems Center, Gunter AFB, Alabama. He earned a Master of Business Administration degree from Auburn University at Montgomery in 1993. He entered the Air Force Institute of Technology in May 1993 to pursue a Master of Science degree in Computer Science.

Permanent address: Montgomery, AL 36116

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Master's Thesis

4. TITLE AND SUBTITLE
USING OBJECT-ORIENTED DATABASE TECHNOLOGY TO DEVELOP A MULTIPLE DOMAIN CAPABILITY FOR DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEMS

5. FUNDING NUMBERS

6. AUTHOR(S)
Alfred W. Harris, Jr., Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/94D-07

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Wright Laboratories/AAWA-1
Wright-Patterson AFB, OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis describes the design and implementation of a multiple domain capability for a domain-oriented application composition system, named Architect. The research goal was to show how object-oriented database management system (OODBMS) technology can be used to provide simultaneous access to multiple domain-oriented knowledge bases. Since the Architect system was originally designed using the object-oriented paradigm, insertion of OODBMS technology was relatively simple and many of the object-oriented concepts, such as inheritance and aggregation, proved beneficial. Inheritance was used to encapsulate domain knowledge by defining each domain as a subclass of Architect's software architecture. Aggregation was used to allow applications to cross domain boundaries by nesting components from multiple domains in an application. To validate this approach, domain extensions to two existing domain models were implemented to make the domains compatible in a multiple domain environment, and applications containing objects from both the logic circuits and digital signal processing domains were successfully developed. One of the primary benefits of this research is the potential for greater reuse of objects. To satisfy new requirements, domain engineers can now search for and access objects from other domains as an alternative to implementing them in their own domains.

14. SUBJECT TERMS	15. NUMBER OF PAGES
object-oriented database management system, aggregation, inheritance, domain-oriented application composition system	117
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL